Escola Politècnica Superior
d'Enginyeria de Vilanova i la Geltrú

UNIVERSITAT POLITÈCNICA DE CATALUNYA

PROJECTE FI DE CARRERA

TÍTOL: EWMR – Engine WorkFlow Models & Rules

AUTOR: Tomás Santaeufemia Rodríguez

TITULACIÓ: Enginyeria Tècnica en Informàtica de Gestió

DIRECTOR: Jordi Daude

DEPARTAMENT: Llenguatges i Sistemes Informàtics

DATA: Junio 2012

Aquest Projecte té en compte aspectes mediambientals: □ Sí □ No

PROJECTE FI DE CARRERA

RESUM

El proyecto EWMR, pretende lograr como primer objetivo la independencia entre el usuario final que modeliza o genera flujos de trabajo (con tomas de decisión) y los departamentos técnicos que son normalmente los encargados de realizar los desarrollos que propone el usuario.

Para ello, se ha implementado un programa que después de una única instalación, permite al usuario realizar cambios sin que el departamento de sistemas tenga que intervenir. Gracias a esta independencia, se consiguen acelerar los tiempos de puesta en producción de los cambios o los nuevos flujos diseñados (ya que no se requiere pasar por pruebas en diferentes entornos) y en consiguiente un ahorro de costes. También otro punto a tener muy en cuenta, es que se consigue no trasladar fuera de los departamentos que toman las decisiones el conocimiento que estos poseen, ya que en determinados ámbitos esto puede llegar a ser un proceso crítico.

Analizando las aplicaciones existentes en el mercado, podemos llegar a encontrarnos programas que en parte pretenden cubrir todas estas necesidades. La realidad es que en muchas ocasiones poseen características muy limitadas y no suplen al 100% todas las necesidades.

En el proyecto, se ha desarrollado un aplicativo para el entorno PC, para poder probar los flujos generados y otro para el entorno HOST, entorno de producción. Siendo el de entorno HOST conectado a BBDD, para que a través de la misma se pueda obtener parte de la información que se requiere para la ejecución.

EWMR al tratarse de un intérprete y no un código a compilar, consigue cubrir el principal hito que es la independencia. Adicionalmente se le ha dotado de más funcionalidades de las que poseen programas similares, como son:

- Número ilimitado de operaciones dentro de reglas y bucles.
- Las interfaces de entrada y salida de datos se pueden alterar sin que por ello se deba variar el programa. Son totalmente dinámicas.
- Permite la realización de flujos complejos (no lineales).
- Expresiones de cálculo complejas y con la máxima precisión.
- Se pueden llegar a incorporar nuevas funciones.
- El CORE del sistema PC y HOST es el mismo, por lo cual nos aporta mayor fiabilidad.

Paraules clau:

COBOL	HOST	DB2	FLUJOS
UAT	SAS	DYD	

Contenido

1. Introducción

Desde que empecé los estudios de Informática de Gestión, he estado trabajando en dos empresas: la primera una consultora que se dedicaba a realizar proyectos informáticos para Cajas de Ahorro y Banca; y actualmente estoy desarrollando mi carrera profesional en una entidad bancaria. Gracias a esto he podido ver los dos mundos: el de proveedor de servicios y el de cliente que los demanda.

En la consultora aprendí a desarrollar diferentes aplicativos y en muy variados lenguajes (VB, VBA, JAVA, C, C++, COBOL, NATSTAR, NSDK, VBNET, C#, .NET). En mi última etapa como analista, pasé desde la programación de los aplicativos a luego realizar estimaciones de tiempos y costes de desarrollo, así como la organización de mi propio equipo de trabajo.

En la etapa actual como cliente, me he encontrado con diversas herramientas de proveedores de servicios que en muchas ocasiones eran productos similares a los que desarrollaba en mi anterior trabajo. Esto me ha servido, conociendo las tecnologías empleadas, a poder estimar mejor los tiempos de desarrollo de evolutivos que pedimos o de nuevas funcionalidades y así como poder exigir más a los que ahora son mis proveedores.

Gracias al trabajo que desempeño, he podido comprobar, que cada vez más las empresas se encuentran con mucha información histórica almacenada en BBDD de Host. Esta información histórica, constituye un gran valor para las empresas ya que a través de modelos estadísticos y flujos de trabajo, sabiendo que es lo que ha pasado en situaciones anteriores, les puede ayudar a predecir el comportamiento de operaciones similares en el futuro, a través de aplicaciones que se denominan de inteligencia artificial.

La implementación de estos modelos estadísticos, sale normalmente de la realización de estudios a través de los programas como SPSS, R o SAS por parte de equipos estadísticos.

La implementación de los modelos estadísticos se suele realizar a través de equipos de diseño y desarrollo que pertenecen a la empresa que posee la información o bien contratando a factorías de desarrollo externas. Cada vez, es más frecuente, que las empresas decidan externalizar los desarrollos, por temas de costes, ya que en muchas ocasiones no es posible soportar un departamento de desarrollo dentro de la propia empresa.

Sea la factoría externa o el equipo de desarrollo interno, nos encontramos que estos equipos, en muchas ocasiones solo cuentan con información muy

técnica, funcionales o escuetos documentos con los cuales es difícil realizar la implementación, con lo que los desarrollos acaban siendo largos y tediosos, ocasionando que en muchas ocasiones se deban realizar muchos correctivos hasta conseguir afinar el producto. Estos problemas no solamente vienen motivados por una deficiente documentación, sino porque los equipos de desarrollo no poseen la misma formación que los equipos estadísticos, ni conocimientos del objetivo que se pretende realmente alcanzar. Dicho esto, en ocasiones podemos llegar a decir que no son más que meros transcriptores del código que ha sido generado desde los equipos estadísticos.

A la hora de generar nuevos productos, no solamente las empresas o entidades se encuentran con los problemas antes mencionados, también se encuentran que hay cierta información crítica que no puede o no debe salir de los departamentos de ámbitos de decisión. Por lo que en ocasiones no es factible la contratación de factorías externas.

Con la problemática anterior, no cabe más que la opción de tener un equipo propio de desarrollo, del cual siempre se acaba teniendo una gran dependencia. Ya que si se desea acometer cualquier tipo de cambio o funcionalidad, son solamente ellos quienes los pueden realizar.

Las empresas están demandando cada vez con más frecuencia, mecanismos que les permitan desligar procesos críticos, por confidencialidad o porque requieren de ágiles implantaciones de productos de los departamentos de desarrollo.

Para subsanar esta gran problemática, en el mercado están saliendo una serie de aplicativos que pretenden eliminar esta dependencia que tienen las empresas de los equipos de desarrollo (Por ejemplo: "Strategy Manager" de la empresa Experian Scorex).

El proyecto que se expone a continuación, pretende ser parte de la solución de un producto que es altamente demandado por las entidades y cajas. Además de que ya ha tenido una aplicación real, ya que está instalado en una asociación de cajas y bancos llamada ATCA (ubicada en Zaragoza), actualmente adquirida la infraestructura tecnológica por IBM.

El desarrollo integro del aplicativo EWMR ha sido elaborado teniendo en cuenta estas necesidades, tanto el diseño como la programación ha sido elaborado por mí. Puedo dar gracias a que en esos momentos estaba estudiando una de las asignaturas que me parecen mas interesantes de la carrera como EDAL y asignaturas de bases de datos con las cuales me fue viable la construcción de todas las estructuras de datos para que el proyecto saliese adelante.

2. Objetivo

El objetivo de este proyecto consiste en crear una herramienta, que de ahora en adelante la llamaremos motor, que sea capaz de interpretar de forma "autónoma" las instrucciones que se programen: ya sean flujos de trabajo (Workflows) o reglas y operaciones matemáticas.

El proyecto EWMR – Engine WorkFlow Models & Rules, tiene como principal objetivo lograr la independencia entre el usuario final que modeliza, diseña o genera flujos de trabajo (con toma de decisiones) y los departamentos técnicos que son normalmente los encargados de realizar los desarrollos que propone el usuario y posteriormente su implantación o instalación.

Se pretende por ello crear una librería o estructura que después de realizar una única instalación, permita de forma automática al usuario realizar los cambios de forma ágil. Estos cambios, pueden venir motivados simplemente porque se deseen realizar pruebas sobre los datos existentes o ficticios, o porque directamente se deseen alterar las condiciones que existen en el entorno productivo tras haber visto que una política, modelo o flujo no cumple con los resultados esperados.

De esta forma se consigue también acelerar los tiempos de implantación y se reducen los costes, ya que se evita tener que pasar por la realización de instalaciones y posteriormente pruebas en todos los entornos previos a producción, como también las UAT (Pruebas de Aceptación del Usuario).

Los siguientes puntos son las principales premisas e ideas con las que partí a la hora de realizar este programa.

- Instrucciones sencillas: la programación de instrucciones que requiere el intérprete deben ser lo más sencillas posibles, para ello hemos usado un metalenguaje. De esta forma, evitamos que el usuario requiera de conocimientos informáticos.

- Mantenimiento cero: esta también es una premisa indispensable, una vez instalado, el software no debe requerir de nuevas instalaciones, a pesar de que se varíe el código a ejecutar. Por lo cual, visto desde el punto de vista medioambiental podríamos decir que reducimos las emisiones de CO2 derivadas de los desplazamientos que se tendrían que realizar para hacer ajustes o implantar nuevas versiones.

- Reducción en la documentación que se tendría que trasladar a los equipos de DyD (Diseño y Desarrollo) si se tuviese que realizar la programación de los modelos implementados. Con lo cual podemos decir que el aplicativo en este aspecto también respeta el medioambiente (reducción de documentación extensa en papel). Para realizar la instalación del aplicativo, se requiere solamente de un manual muy breve, ya que los módulos son muy sencillos, cosa que también aporta una rápida instalación.

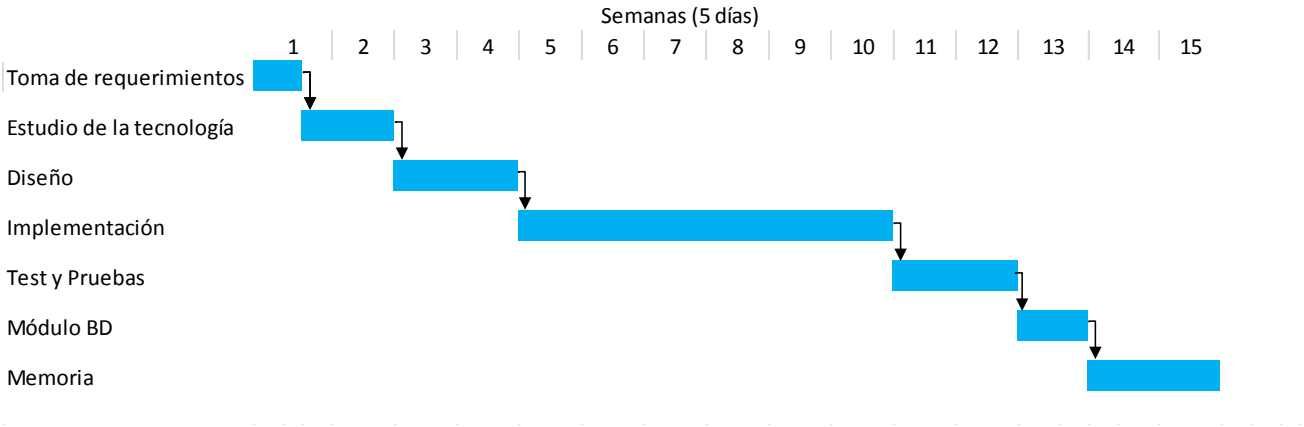
- Mejorar las funcionalidades técnicas con respecto a los productos de la competencia.

Se pretende llevar a cabo el PFC, con los conocimientos que se han adquirido en muchas de las asignaturas presentes durante los estudios de Ingeniería Técnica en Informática de Gestión, en su gran mayoría las que pertenecen al departamento de LSI como pueden ser EDAL, FBDA o CGES. Ya que se van a requerir para la construcción, estructuras de datos, métodos de ordenación y el uso de instrucciones de BBDD y así como la planificación de proyectos.

3. Planificación y Costes

3.1 Planificación

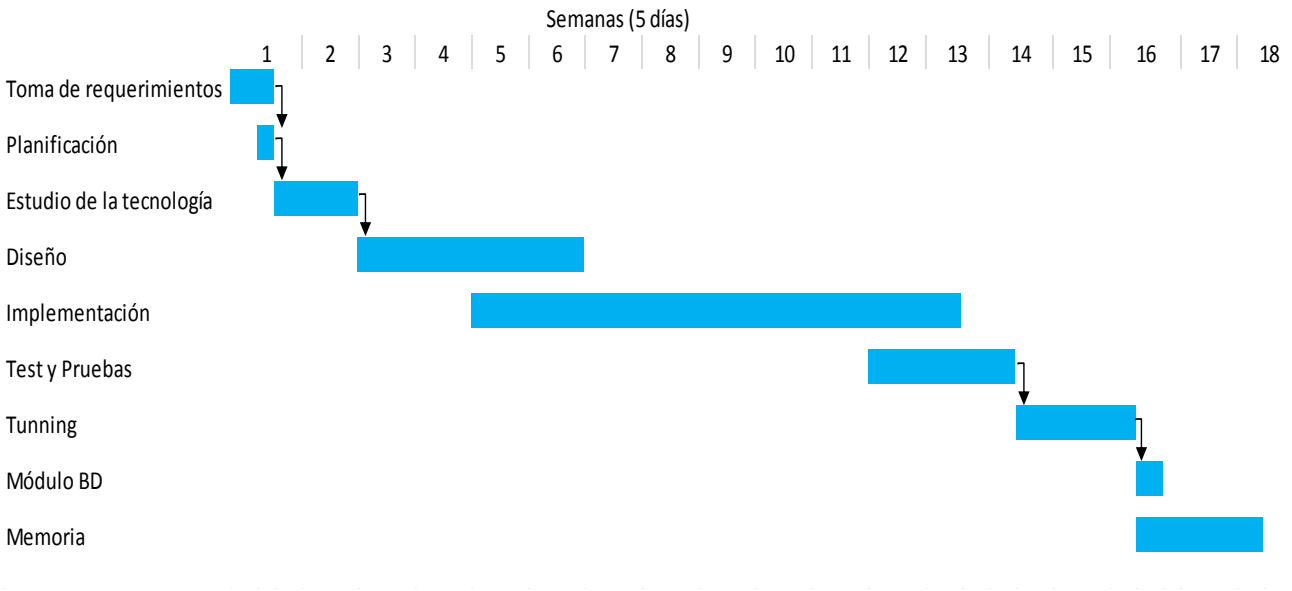
La estimación inicial de este proyecto es la que se muestra en la siguiente imagen:



Como se puede observar inicialmente el proyecto duraba 15 semanas (75 días), pero una vez acabada la fase de desarrollo y debido a problemas de rendimiento, decidí realizar un rediseño total del aplicativo.

Durante el desarrollo inicial del proyecto, busqué emular las funcionalidades que había observado en otros programas de similares características. Me encontré con un grave problema de rendimiento, después de la fase inicial de diseño e implementación, y decidí que debía corregir esa deficiencia que planteaba la programación. A medida que optimizaba el código, me percaté que era posible implementar nuevas funcionalidades que no tienen programas como éste. Finalmente, puedo decir que acabé reimplementando o rehaciendo todo el trabajo que se había realizado inicialmente, ya que cree muchas funciones que simplificaban el código y lo hacían más inteligible y fácil de mantener.

La planificación final del proyecto tras la reimplementación es la que se muestra en la siguiente imagen.



Como se puede observar la duración total del proyecto se ha llevado a cabo en 87 días. Con un desfase entre la planificación inicial y final de 12 días, un 16% más de tiempo empleado.

A pesar de mi experiencia en gestión de proyectos informáticos y equipos de desarrollo, y tal como nos decían los profesores que tuve en asignaturas de informática de gestión relacionadas con la estimación de proyectos (como la asignatura de CGES), la estimación de proyectos informáticos no es trivial. "En la construcción de una casa podemos saber casi a ciencia cierta cuando finalizará, pero con el desarrollo de software es una tarea muy compleja". Aunque poseas mucha experiencia, se suelen dar incidencias o problemas que hacen que finalmente se incurran en retrasos.

Normalmente, sobre las estimaciones o planificaciones que se realizan, se suele usar factores multiplicadores. De forma que sobre las estimaciones iniciales que hemos calculado para cada una de las tareas, nos permitan tener una horquilla o margen de maniobra por si surgen factores que no hemos contemplado inicialmente y hacen que se alarguen los tiempos y por tanto los costes. Así se evita que en muchos casos, los proyectos informáticos sean deficitarios para quien los desarrolla, ya que en muchas ocasiones no se pueden trasladar estas desviaciones al cliente final.

No obstante, estos márgenes a veces se quedan cortos y por ello cuando surge algún tipo de problema que hace que se alarguen los tiempos de desarrollo o análisis hay que emplear diferentes opciones para intentar mitigar estos efectos, entre ellas:

- dedicar más horas en la jornada de trabajo ya establecida (horas extras)
- tener más gente dedicada al proyecto, esto puede ser o bien en momentos puntuales o durante toda la duración del proyecto. Lo más normal es la primera opción, ya que la segunda suele estar asociada a planificaciones, las cuales se sabe de antemano que no se pueden cumplir en los tiempos requeridos desde un inicio del proyecto. La primera se utiliza más cuando nos hemos encontrado con algún tipo de contratiempo.
- la última consistiría en subcontratar tareas específicas, o lo que es lo mismo el outsourcing.

Hay que tener en cuenta que las dos últimas opciones, contratación de más personal o externalización, conllevarán fases de análisis y aprendizaje por parte de las personas que realicen las tareas. Por lo que en muchas ocasiones conseguimos reducir los tiempos, pero no sin dedicar buena parte de los esfuerzos a que adquieran los conocimientos oportunos para llevar a cabo el trabajo.

En el caso particular de este proyecto, no ha sido posible realizar ninguna de las 3 opciones que he planteado. Las dos últimas están claramente descartadas, ya que estamos hablando de un proyecto de final de carrera y como mucho podría haber propuesto realizar el proyecto conjuntamente con otra persona, pero no es el caso. La primera opción tampoco no fue muy factible, dado que tuve que compaginar el proyecto con mi vida laboral o profesional. No obstante sí que me ha sido posible el trabajar fines de semana y festivos, ya que entre el trabajo y el proyecto no me quedaban horas en el día para poder abordar todos los desarrollos.

3.2 Costes

Los costes del proyecto los he dividido principalmente teniendo en cuenta tres posibles perfiles, aunque como ya he comentado, la totalidad del proyecto lo he realizado yo.

Los perfiles son los que se muestran a continuación:

- Jefe de proyecto
- Analista
- Desarrollador o programador.

En la faceta de jefe de proyecto se han imputado aquellas tareas que son más puramente de gestión, como son principalmente las tareas de toma de requerimientos y la planificación.

En la de analista, se ha encargado de realizar el análisis, diseño, supervisión de los desarrollos y así como de elaborar la documentación.

Por último tenemos al desarrollador o programador, que es quien llevará a cabo todos los desarrollos, la instalación del producto y así como la puesta a punto del programa.

A continuación se puede ver una tabla que muestra los costes por hora de trabajo que hemos empleado para cada uno de los diferentes perfiles que tenemos definidos en el proyecto:

Perfil	Coste hora
Jefe Proyecto	100 €
Analista	40 €
Programador	20€

Según la tabla anterior, y entrando más en detalle con el número de horas planificadas para cada perfil y las que luego finalmente se realizaron, se han elaborado las dos planificaciones detalladas que se pueden consultar en las imágenes que se muestra a continuación (Planificación Inicial y Planificación Final). Como consideraciones generales de la planificación, se han aplicado diferentes colores para indicar qué perfil es al que le atañe su elaboración.

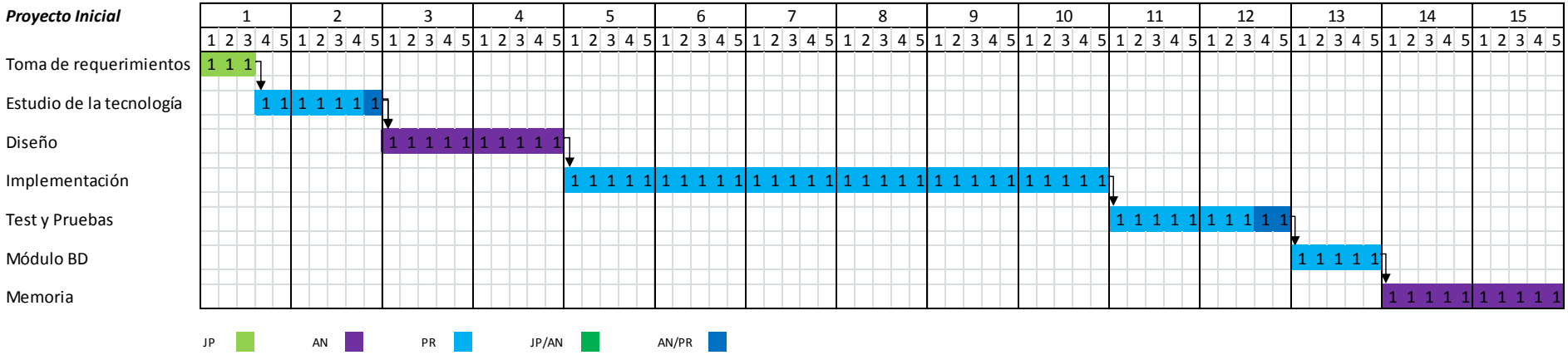
Estos colores en su alta mayoría se corresponderán con los 3 perfiles definidos, no obstante, se han incorporado 2 colores adicionales que nos permiten contemplar aquellas tareas que en algún momento determinado deban ser compaginadas por dos funciones a la vez, como pueden ser:

- Estudio de la tecnología – ya que a pesar de que hemos asignado esta tarea al programador, la toma de decisión final de cual emplear, le corresponde al analista que la tomará en función de lo que le comente el programador.

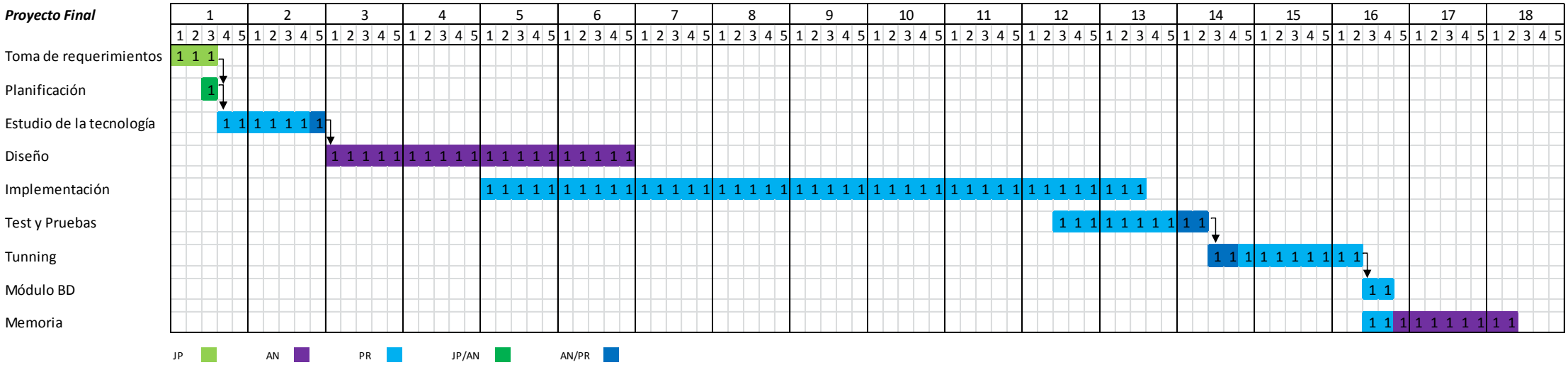
- Test y Pruebas – Esta es otra tarea que principalmente le corresponde al programador, pero es la función del analista la que se tiene que encargar de cerciorarse de que el producto final funciona de forma correcta y sigue las especificaciones o requerimientos que se acordaron con el cliente y el jefe de proyecto.

- Planificación – Tarea exclusiva del Jefe de Proyecto, pero la cual debe consensuar con el analista para verificar que se cumplan los plazos y términos acordados.

Proyecto Inicial



Proyecto Final



En primer lugar vamos a mostrar el detalle de número de horas y el coste de la estimación inicial que se realizo del proyecto:

Perfil	Coste hora	Nº Días	horas	Coste por perfil
Jefe Proyecto	100 €	3	24	2.400 €
Analista	40 €	20,5	164	6.560 €
Programador	20 €	50,5	404	8.080 €
Coste Total	28,8 €	74	592	17.040 €

A continuación se muestran los costes finales del proyecto:

Perfil	Coste hora	Nº Días	horas	Coste por perfil
Jefe Proyecto	100 €	2,5	20	2.000 €
Analista	40 €	26	208	8.320 €
Programador	20 €	58,5	468	9.360 €
Coste Total	28,3 €	87	696	19.680 €

Como se puede apreciar el coste medio del proyecto (coste total/horas totales) ha bajado con respecto a la estimación inicial de 28,8€ a 28,3€, no obstante se ha incrementado el coste del proyecto en 2.640€ y 104 horas más de trabajo. Esto principalmente ha sido debido a que se han tenido que emplear más horas de analista y sobretodo de programador, ya que son más baratas las horas.

Se ha reducido en 4 horas (media jornada) las horas del Jefe de Proyecto, ya que lo lógico es que la planificación se realice de manera conjunta con el analista, tarea que inicialmente no se había planificado en el diagrama Gant.

Las horas del analista se han incrementado en 44 horas (una semana y media jornada más). Principalmente, esta desviación se produjo debido a que cuando se comenzó la implementación, surgieron una serie de dudas conceptuales que originaron que se incrementasen los tiempos de diseño que se compaginaban en paralelo con la implementación. Adicionalmente, y debido al problema de rendimiento que se ha mencionado en este documento, se tuvieron que buscar técnicas de optimización del código COBOL, usando tipos de datos más eficientes, acotando las estructuras empleadas en la implementación, como también usando otras técnicas de construcción de estructuras de datos y técnicas de ordenación vistas en la asignatura de EDAL. Se tuvieron también que revisar, las estructuras recursivas más empleadas y optimizar el código para que estas tuviesen menos instrucciones y fuesen más eficientes.
Por otra parte, se pudieron reducir las horas empleadas en la realización de la documentación (16 horas).

Por parte del programador o desarrollador, se ampliaron las horas en la fase inicial de la implementación, ya que se tuvieron que compaginar con el diseño. Adicionalmente, se tuvo que compaginar la tarea final de implementación con la

de Test y Pruebas. Esto fue en parte necesario debido al problema de rendimiento, ya que se tuvieron que realizar muchas pruebas mientras se programaba, para así validar que se conseguía afinar el "Tunning" de la aplicación (tanto en uso de memoria como de proceso o velocidad).

Por otra parte, se consiguió realizar en un tiempo record el módulo de BD, ya que gracias a las técnicas de encapsulamiento de funciones y de procesos, se consiguió reaprovechar el módulo de lectura desde ficheros para adaptarlo a que se comunicase con la base de datos DB2 de la cual se extraía la información. En este punto se pasó de 40 horas a 4. Lo que permitió colaborar en la tarea de documentación, tarea que podríamos decir era integra del analista.

4. Análisis

La fase de análisis es una de las más importantes, ya que es precisamente donde se debe de llevar a cabo la toma de decisiones sobre qué se quiere implementar (llevando a cabo la toma de requerimientos), qué tecnología se va a usar, y cuál va a ser su aplicación práctica.

4.1 Análisis de herramientas similares existentes en el Mercado

En este punto, se pretenden recoger las características que suelen contemplar programas o aplicaciones similares al que es objeto este PFC.

Esta tipología de programas suelen tener características muy básicas;

- Interpretación de reglas básicas de lógica matemática sencillas, bajo el uso de metalenguaje. Ejemplo:

```
SI A=B
   ENTONCES C
SINO
   ENTONCES D
FINSI
```

- Interpretación de instrucciones o cálculos matemáticos simples. Ejemplo:

```
A = 2;
B = 8 + A;
C = B * 9;
D = C / 2;
```

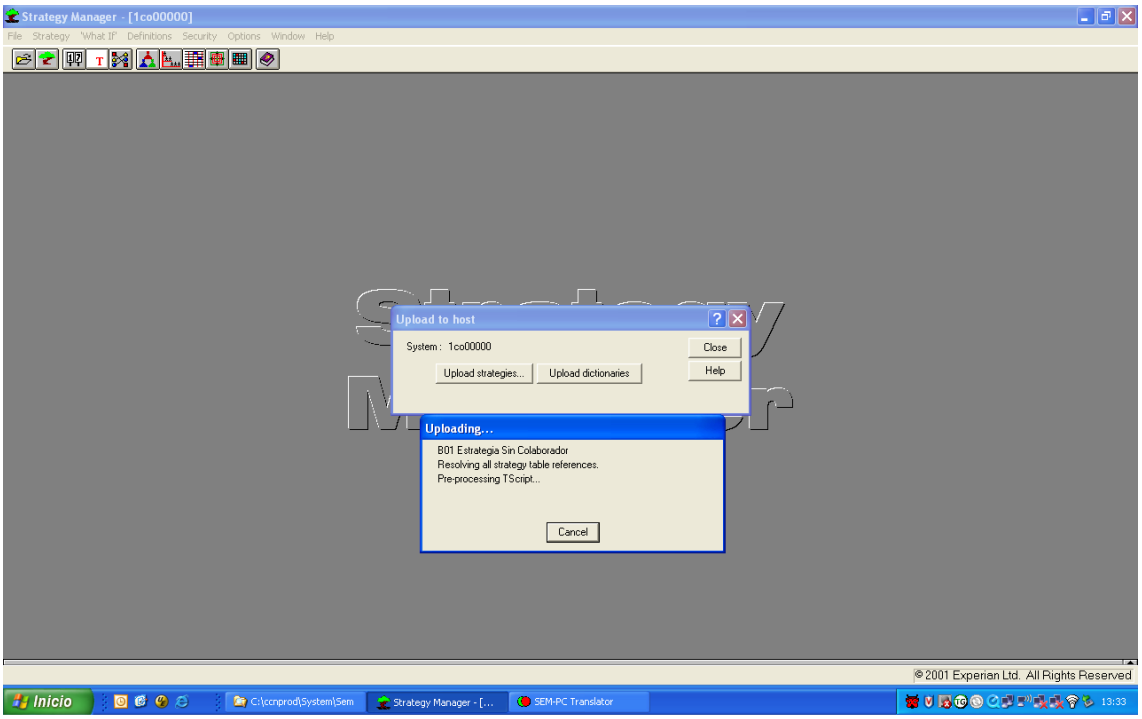
- Definición de estructuras de datos fijas, de entrada para aprovisionamiento del aplicativo de toma de decisión y de salida como resultados.

Con estas 3 características básicas, se pueden llegar a trasladar gran parte de los desarrollos, que realizan las áreas de DyD (Diseño y Desarrollo) hacia los usuarios que analizan la información. A pesar de ello, al ser el pseudocódigo tan trivial, a su vez lo hace muy complicado de uso, ya que si queremos trasladar operaciones más complejas, nos obligará a usar muchas condiciones en las reglas y nos obligará a la larga a requerir de mucho código repetido.

Estos programas que permiten el uso de un metalenguaje, de altísimo nivel, de alguna manera tienen que interpretar esa codificación propia y transformarla a través de un software propietario, para que en definitiva genere un código que puede ser comprendido por la máquina. En muchos casos podríamos decir que se trata de un pre-compilador que es el que se encarga de traducir la información de reglas y cálculos matemáticos para que sean interpretados por los ordenadores o Host en formato máquina.

Los pre-compiladores que usan este tipo de programas son de manejo muy sencillo, ya que no requieren de parámetros de configuración como ocurre a la hora de compilar con paquetes como JAVA, C++, u otros lenguajes.

Ejemplo de pre-compilado, de este tipo de herramientas:



En la mayoría de los casos, estos pre-compiladores permiten 2 cosas: validar que no existan errores en el flujo o condiciones que se han introducido, y por otra parte, permiten que se puedan realizar pruebas en el sistema en el cual se han pre-compilado.

¿Y en un Host o MainFrame[1], cómo se realiza el compilado? En las grandes empresas o entidades no es posible que un usuario tenga, ni pueda realizar modificaciones de paquetes informáticos que estén ya alojados en servidores, Host o Mainframe. Los problemas con los que nos solemos encontrar son principalmente por temas de seguridad, cumplimiento de normativas internas (calendarios de implantaciones), desconocimiento de procesos informáticos

[1] Mainframe, computadora central para el procesamiento masivo de datos

avanzados (montaje y ejecución de JCL[2], etc.), por lo que a pesar de que pudiéramos perder gran parte de la dependencia que tenemos de los equipos informáticos, gracias a este tipo de aplicativos, no es posible que después podamos realizar un cambio de forma ágil.

Principalmente, cuando queramos realizar un cambio, una vez que ya se ha implantado una versión o programa, nuevamente tendremos que ponernos en contacto con los equipos de implantación para que realicen el compilado, instalación y finalmente las pruebas, para que así luego, los usuarios puedan probarlo con la información de entornos productivos (UAT).

En pocas palabras, con este tipo de aplicaciones, desaparece la dependencia de otras áreas para corregir o modificar los programas, pero no para decidir cuando queremos aplicarlos, ya que normalmente se escapa del ámbito de los usuarios y no está en nuestra mano.

Caso real

A continuación expongo un ejemplo real de programa de reglas:

En este caso nos vamos a basar en el programa de la casa Experian. Esta casa posee un aplicativo, el cual, permite "programar" o "modelizar" las reglas que queramos desarrollar desde las áreas usuarias. Una vez que se ha realizado el desarrollo, a través de un aplicativo de nivel de usuario, podemos "pre-compilar" el software que luego será planificado para ejecutarse en el Host. Este programa "pre-compilado", no es más que un par de ficheros planos, que contienen las instrucciones que luego deberán ser mandados a los departamentos informáticos, para que a través de otro aplicativo desarrollado por la misma Experian, que está instalado en el Host, se haga la traducción de los ficheros planos a lenguajes que el Host pueda interpretar como son COBOL o Java.

Características principales:

---

[2] (acrónimo de **Job Control Language**, en español **Lenguaje de Control de Trabajos**) conjunto de especificaciones de morfología y sintaxis requeridas para la redacción de instrucciones de ejecución de programas informáticos por parte del sistema operativo de un equipo informático. Este lenguaje se usa en los Ordenadores Centrales (Mainframes) y es específico para cada sistema operativo.

Las instrucciones (también llamadas "pasos" o "sentencias") del JCL son declaraciones u órdenes con las que se indica al sistema operativo qué tareas debe realizar, en qué secuencia han de ejecutarse y en qué periféricos están ubicados los ficheros de datos (de entrada y/o de salida) que requieren dichas tareas.

- Puede trabajar con hasta 3 niveles de reglas anidadas y no permite más de 250 instrucciones dentro de una regla. Ejemplo:

```
SI A = B ENTONCES
   SI C = D ENTONCES
         SI E=F ENTONCES
               …….
         FINSI
   SINO SI A=D ENTONCES
         SI A=C ENTONCES
               …….
         FINSI
   FINSI
ENDSI
```

- No puede trabajar con muchas instrucciones de cálculo a la vez, en una misma sentencia. Ejemplo:

  Si quisiéramos representar la siguiente expresión:

  A = EXP(((B*C) +(D*J/H))-(1*24))

  se tendría que escribir de la siguiente forma:

```
A1 = ((B*C) +(D*J/H))
A2 = A1 – (1*24=
A = EXP(A2)
```

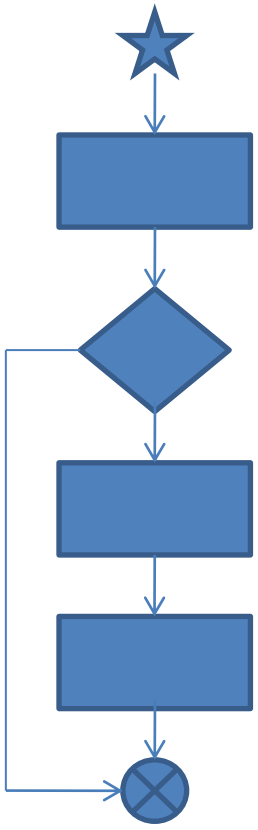
  Por lo que también implica el uso de más variables temporales.

- No permite la "definición" de nuevas formulas matemáticas que no estén ya implementadas. Se pueden construir dentro del pseudocódigo, pero para ser usadas, se deben de construir tantas veces como se deseen usar.

- No permite el uso de iteraciones, por lo cual si queremos hacer algún trozo de código que se repita "n" veces lo tendremos que escribir esas "n" veces, por lo cual el código se hace más extenso y no legible.

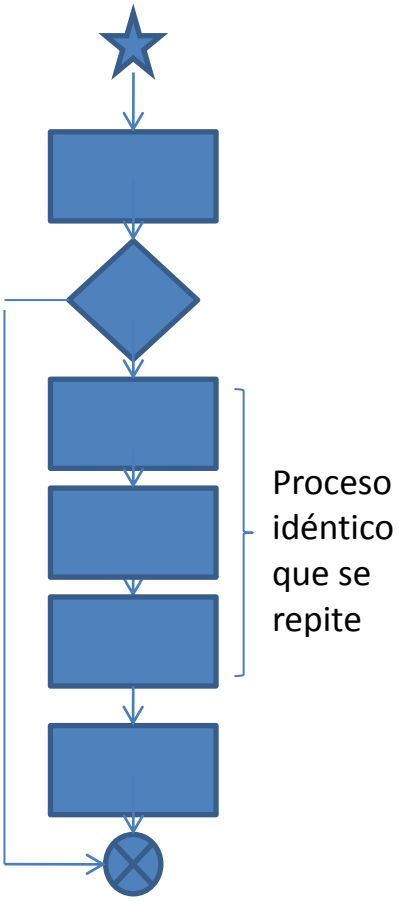
- Solo permite trabajar con flujos lineales. Por lo que no se pueden tener trozos de código compartidos, para que sean reutilizados.

  Ejemplo flujo lineal:

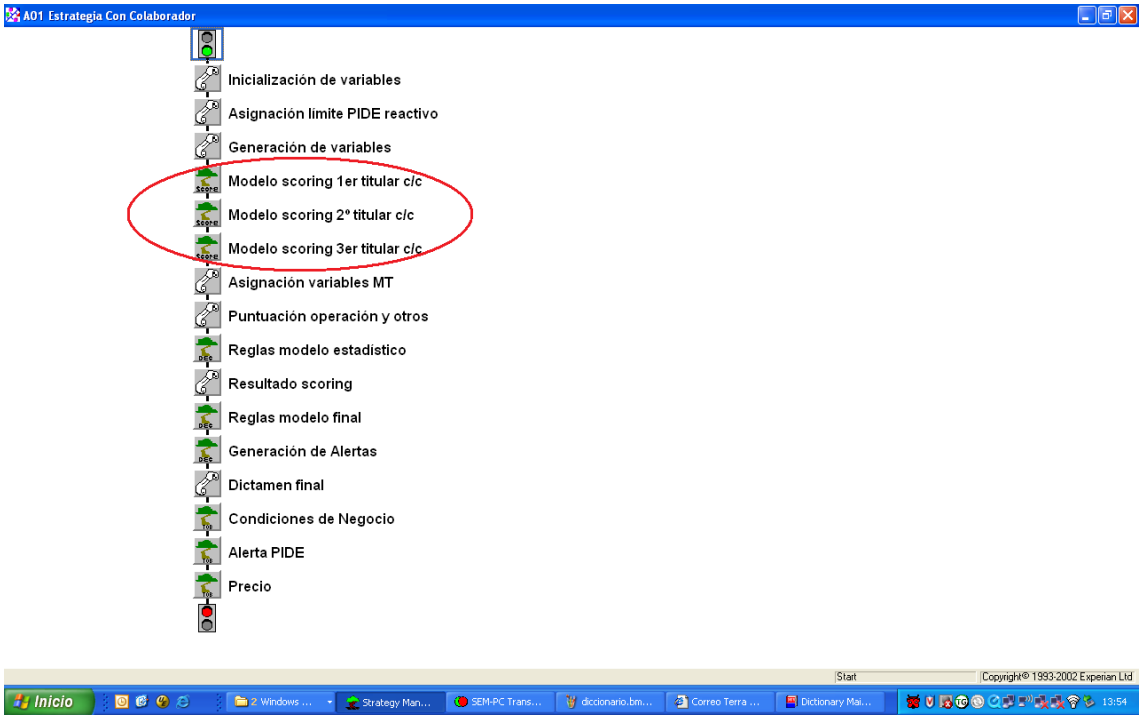


- Para aquellas variables que sean de tratamiento múltiple, aunque el código deba ser igual, se tendrá que escribir "n" veces en función de las veces que se quiera contemplar, por lo cual se limita también el número de veces que queremos evaluar, ya que vendrá determinado por el número de bloques de las variables de entrada y de salida.
  Ejemplo: en caso de que en una operación existan varios intervinientes, se tendrá que construir un módulo adhoc para cada uno de ellos, aunque el tratamiento sea el mismo.

Ejemplo flujo lineal:



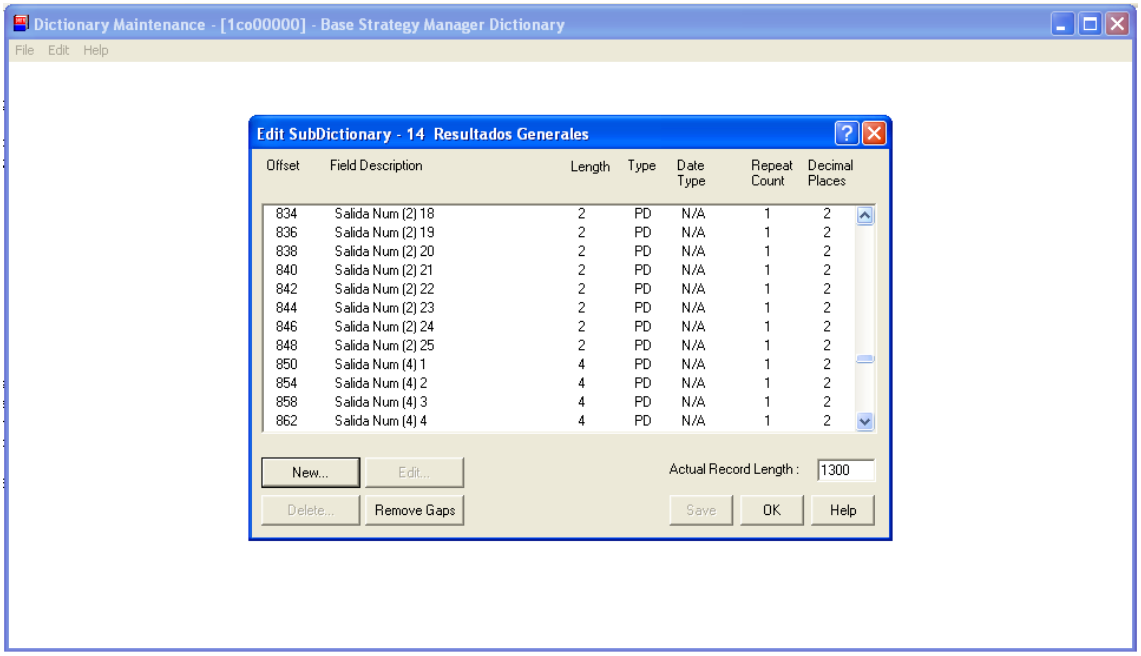
Ejemplo real de flujo lineal (en color rojo, módulo repetido para poder gestionar la multiplicidad de información):



- Los cálculos se realizan sobre las variables definidas, con formato especifico, lo que en muchas ocasiones provoca desbordamientos de la parte entera, o errores en la precisión decimal ya que no se balancea la coma. Esto provoca en los cálculos complejos mucha imprecisión, y por tanto, que varíe mucho el resultado final.

- Todas las variables tienen que estar previamente definidas. Es necesario por tanto, que se generen variables adicionales para su uso futuro, ya que luego si se requiriese de su inclusión, genera unos costos muy elevados y muchos problemas técnicos llegando incluso a provocarse errores. Otro inconveniente que se plantea al tener más variables de las necesarias, es que se incurre en mayores consumos de memoria RAM.

Ejemplo de definición de estructura fija con huecos reservados para su uso futuro:



4.2 Análisis de requerimientos

Después de analizar las herramientas similares existentes en el mercado, y ver que aún existe una gran dependencia de los departamentos informáticos, surgió la idea de realizar un motor de flujos, que fuese más dinámico que los de la competencia, y que nos permitiera obtener una total independencia de los departamentos informáticos (una vez que se haya realizado una primera y única instalación).

Al igual que se da en los sistemas de la competencia, se va a implementar una herramienta cliente y otra servidor, o lo que es lo mismo una herramienta para poder realizar pruebas en entorno PC y otra para la realización de pruebas o ejecución en entornos productivos.

La diferencia principal va a consistir en que la herramienta HOST va a obtener parte de la información para su funcionamiento vía BBDD, de esta forma, se consigue lograr esa total independencia, y así de forma automática, se podrán evaluar diferentes flujos en función de los datos que se manden a evaluar y sin que tengan que intervenir los departamentos técnicos.

A parte de cubrir, lo que entendíamos que tenia que ser una necesidad básica, la independencia con las áreas de desarrollo, la aplicación tiene que cubrir todas las deficiencias que hemos detectado en la fase de análisis de herramientas existentes en el mercado. Por ello se ha trabajado arduamente en el desarrollo de un intérprete avanzado de reglas e instrucciones matemáticas complejas.

El motor tiene que cubrir con los siguientes requerimientos adicionales:

- Tiene que poder trabajar con múltiples niveles de reglas anidadas. No tiene que existir ningún tipo de limitación. Ejemplo:

```
SI A = B ENTONCES
   SI C = D ENTONCES
         SI E=F ENTONCES
               …….
         FINSI
   SINO SI A=D ENTONCES
         SI A=C ENTONCES
               …….
         FINSI
   FINSI
ENDSI
```

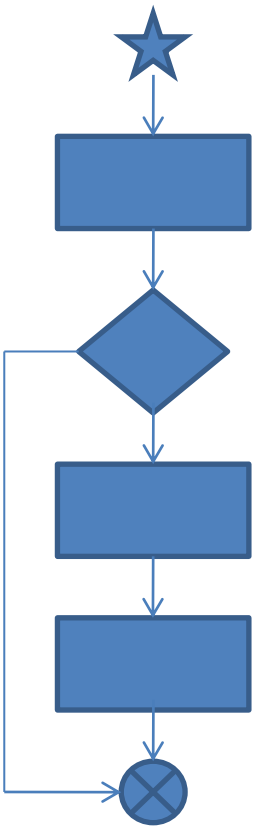
- Tiene que poder trabajar con múltiples instrucciones de cálculo implementadas en una única instrucción.

  Ejemplo:

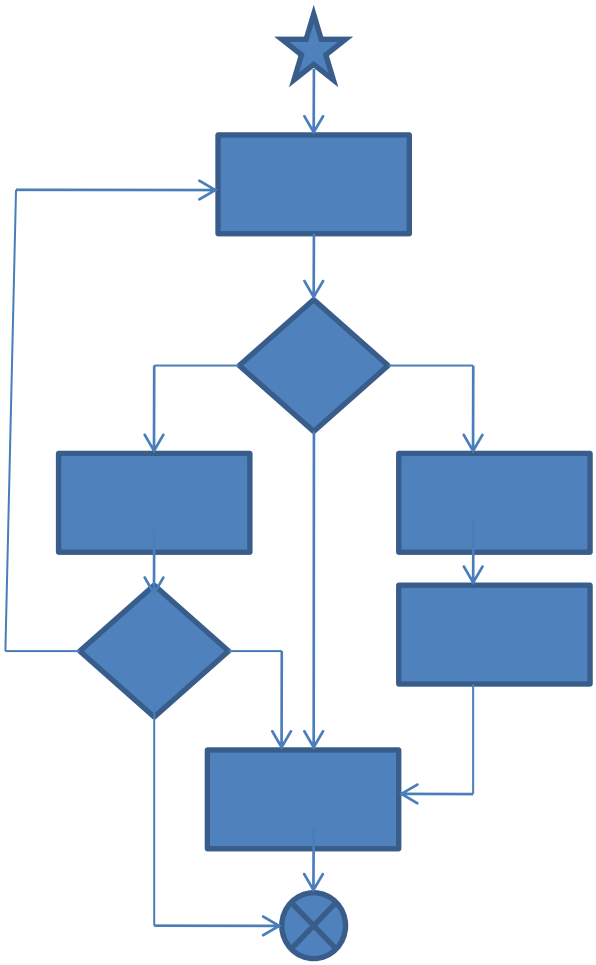
  A = EXP(((B*C) +(D*J/H))-(1*24))

- Tiene que permitir la ejecución de flujos que no sean simples o lineales. Por lo cual, se tienen que poder tener trozos de código compartidos (funciones).

Ejemplo flujo lineal:



Ejemplo flujo no-lineal:

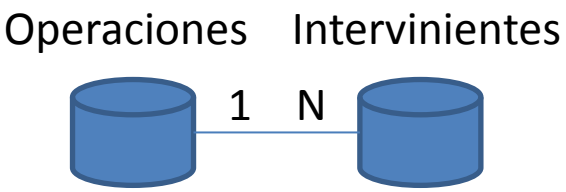


- Se tiene que permitir la "definición" de nuevas fórmulas matemáticas, de forma que no tengamos que programarlas cada vez que queramos hacer uso de ellas.

- Se tienen que poder implementar o construir iteraciones, por lo que podremos realizar tratamiento de procesos de forma múltiple, evitando la réplica de módulos (bucles).

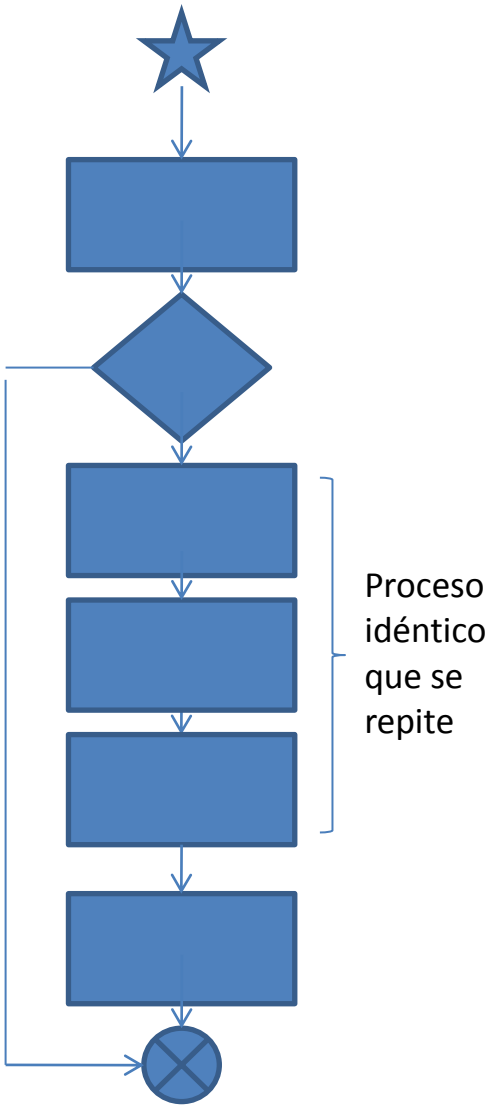
Ejemplo: Imaginemos que tenemos una tabla que contiene todas las solicitudes de préstamos de una entidad bancaria. Y otra tabla que contiene todos los intervinientes de préstamos.

La relación entre ellas sería la siguiente:

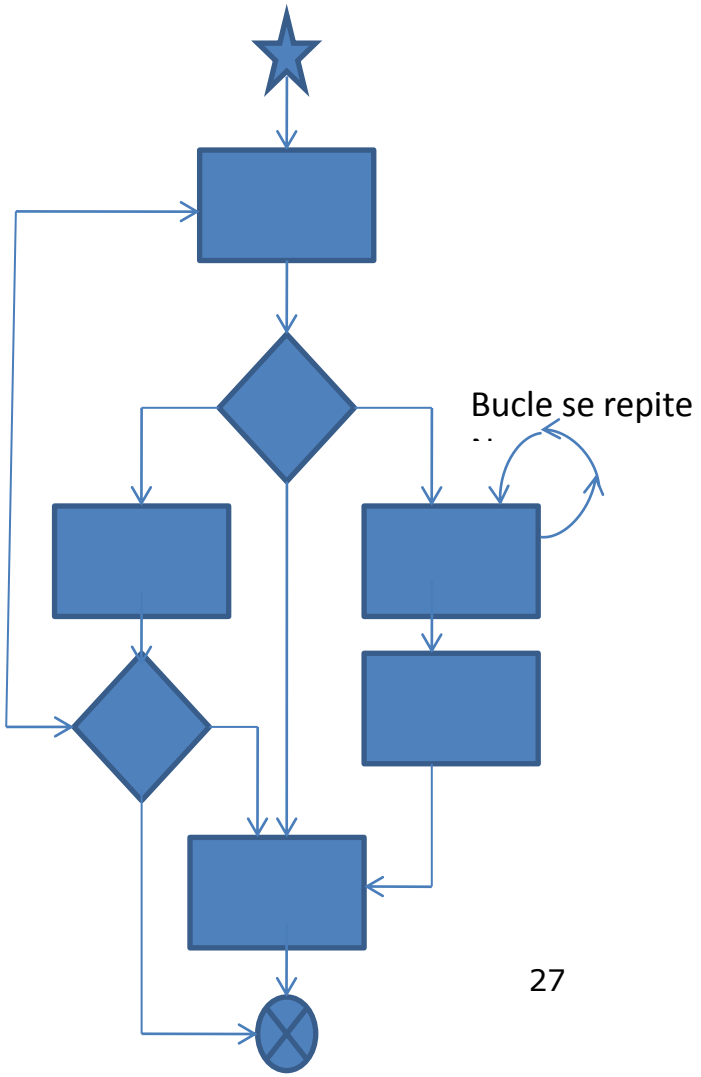


Gracias a la funcionalidad de bucle, dentro de un flujo podremos evaluar la información a nivel de operación y luego, cuando queramos, podremos ejecutar un proceso múltiple, por ejemplo a nivel de interviniente, que nos devolverá la evaluación o resultados de todos los intervinientes. De esta forma, nos evitaremos como se hacen en los flujos lineales, duplicar procesos que son idénticos.

Ejemplo flujo lineal:



Ejemplo Función bucle:



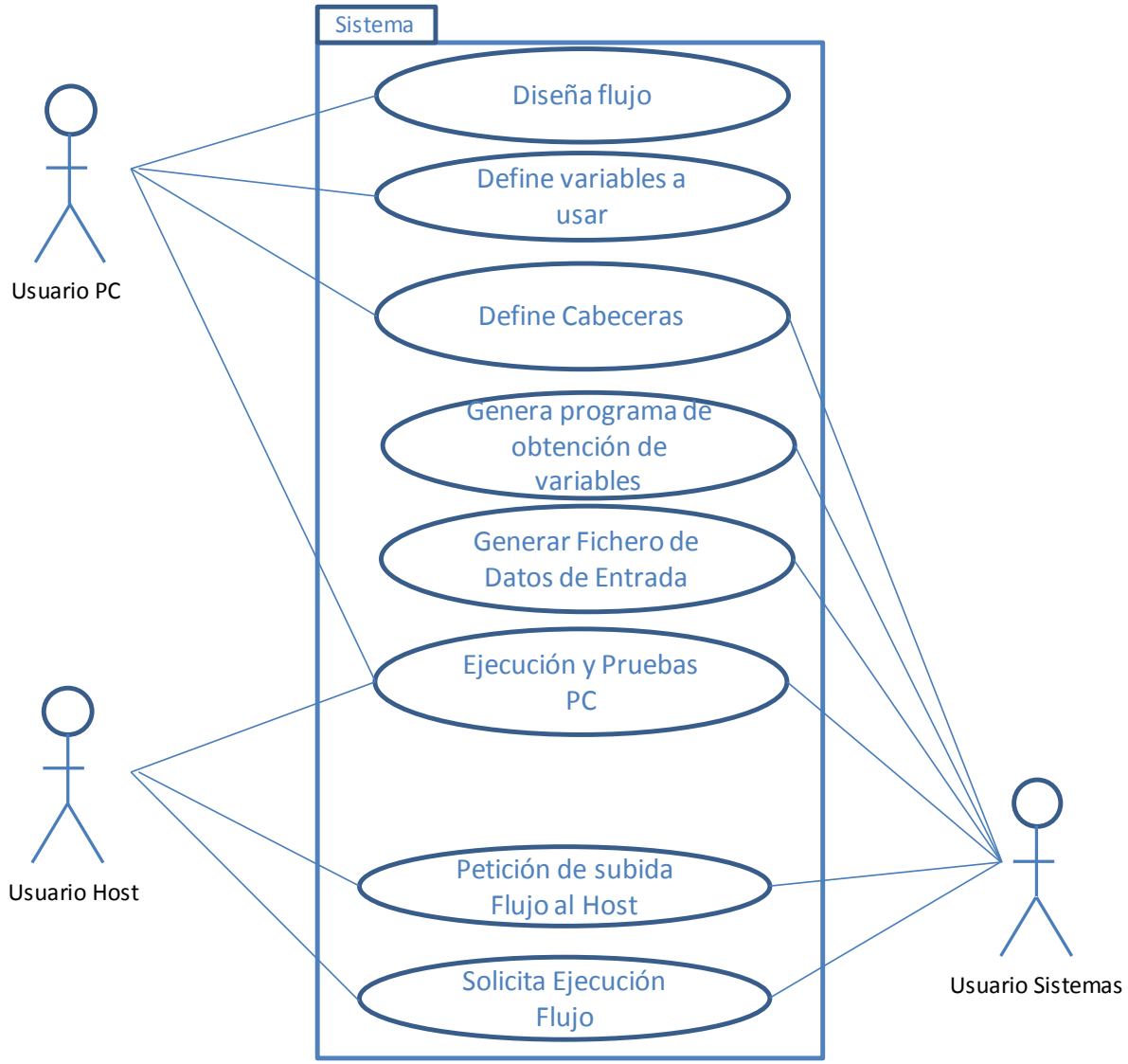
- Los cálculos se realizarán sobre variables con coma flotante, lo que permitirá que los cálculos matemáticos dispongan en todo momento de la máxima precisión posible y que no se provoquen desbordamientos en los cálculos.

- Por último, y no por ello el menos importante, el motor tiene que permitir que la información de entrada y de salida sea dinámica. Con esto queremos decir, que el layout de datos de entrada y de salida dejará de ser estático con lo que lograremos obtener una gran flexibilidad.

Este motor intérprete de estrategias permite que el usuario en todo momento, indique qué campos desea evaluar, y cuáles desea que se retornen por la salida. Esto se hace a través de BBDD o del paso de información por parámetro.

4.3 Casos de Uso

Diagrama del modelo de casos de uso



En el diagrama de los casos de uso se han definido los siguientes actores:

- Usuario PC: Este usuario es la persona experta en diseño de flujos o procesos, y va a ser la encargada de definir el modelo que posteriormente será evaluado. Este usuario, tiene también que definir cuáles son las variables que va a emplear el flujo. En una versión que incluya una interface gráfica podría evaluar el flujo y generar esta interface de forma automática.

- Usuario HOST: Será el encargado de realizar las peticiones de subida de flujos al Host y posteriormente la solicitud de ejecución de ese flujo o cualquier otro que su estado sea activo y también este presente en la

BBDD del HOST. Dará el visto bueno para la subida del flujo al Host, una vez que se haya cercionado de que la ejecución y pruebas del entorno PC hayan sido satisfactorias.

- Usuario Sistemas: Este usuario se corresponderá con un usuario de los equipos técnicos de Diseño y Desarrollo. Se encargará de generar un programa que coja las variables de la BBDD y se transformen a la estructura o interface que requiere la herramienta. Por ello se debe encargar de definir finalmente los ficheros de cabeceras, ya que este usuario es el que tiene los conocimientos de formatos de las variables que ha descargado de la BBDD HOST.

Caso de uso: Diseña Flujo
Actores: Usuario Diseñador PC
Tipo
Descripción: Explicación de cómo diseñar y generar el código EWMR.
Requerimientos
Curso normal de los sucesos
Precondición: El usuario debe conocer en detalle el lenguaje que requiere el programa EWMR
Flujo Normal: 1-El usuario dibuja un flujo, en una herramienta de diseño PC 2-Etiqueta los diferentes objetos implicados, reglas, procesos, bucles, etc.. 3-El usuario crea un esquema en pseudocódigo que se adapte al flujo 4-El usuario completa el esquema, realizando un código detallado 5-El usuario traduce el pseudocódigo al lenguaje del motor
Flujo Alternativo: 5-El usuario revisa que el lenguaje del motor este bien escrito y cumple todos los requisitos.

Caso de uso: Definir variables a usar
Actores: Usuario Diseñador PC, Usuario Sistemas
Tipo
Descripción
Requerimientos
Curso normal de los sucesos
Precondición: Se tiene que haber implementado el código que requiere EWMR.
Flujo Normal: 1-UD_PC, repasa el pseudocódigo e identifica las variables empleadas. 2-UD_PC, una vez identificadas, mira cuál es el código que se le ha asignado a cada una de ellas dentro del código EWMR. 3-UD_PC, comunica a US las variables que quiere usar. 4-US, indica los nombres reales de las variables que tendrá en el fichero CABECERAS. 5-UD_PC genera fichero VARIABLES.
Flujo Alternativo: 2-Se realiza mantenimiento sobre el pseudocódigo y no se alteran las variables de la interfaz, por tanto no se modifica el fichero de VARIABLES.

Caso de uso: Definición Cabeceras
Actores: Usuario Diseñador PC y Usuario Sistemas
Tipo
Descripción
Requerimientos
Curso normal de los sucesos
Precondición: Se tiene que haber definido el fichero VARIABLES
Flujo Normal: 1-UD_PC recomienda al US que variables son susceptibles de usarse en el futuro. 2-US, analiza las posibles variables futuras a emplear. 3-US genera el fichero de cabeceras. 4-US facilita a UD_PC el fichero de cabeceras para la realización de las pruebas.
Flujo Alternativo: 1-No se realizan cambios sobre un fichero de cabeceras anterior.

Caso de uso: Generar Programa de obtención de variables
Actores: Usuario Sistemas
Tipo
Descripción
Requerimientos
Curso normal de los sucesos
Precondición: Haber definido fichero CABECERAS
Flujo Normal: 1-US revisa formatos de las variables del fichero de cabecera con las variables presentes en la BBDD. 2-US realiza programa para adaptar las variables de la BBDD a la interface de CABECERA_IN y CABECERA_OUT que requiere el HOST.

Caso de uso: Generar Fichero Datos de Entrada
Actores: Usuario Sistemas
Tipo
Descripción
Requerimientos
Curso normal de los sucesos
Precondición: Haber definido Fichero Cabeceras y Programa de obtención de Variables
Flujo Normal: 1-US genera fichero de Datos de Entrada

Caso de uso: Ejecución y Pruebas entorno PC
Actores: Usuario Diseñador_PC, Usuario Diseñador Host
Tipo
Descripción
Requerimientos
Curso normal de los sucesos
Precondición: Haber generado el flujo, variables, cabeceras y fichero de datos de entrada
Flujo Normal: 1-US proporciona fichero de datos de entrada a UD_PC. 2-UD_PC prueba la ejecución y realiza pruebas en entorno PC. 3-UD_PC certifica el correcto funcionamiento. 4-Tras OK de UD_PC avisa a UD_HOST para certificación final 5-UD_HOST da el OK para subir a entorno Producción.

Flujo Alternativo: 3-UD_PC no certifica el correcto funcionamiento 4-UD_PC retoca el flujo 5-Se repite el paso 3, si OK realizar pasos 4 y 5 Flujo Normal.

Caso de uso: Petición de Subida Flujo al Host
Actores: Usuario Diseñador Host, Usuario Sistemas
Tipo
Descripción
Requerimientos
Curso normal de los sucesos
Precondición: Finalizar con existo la Ejecución y Pruebas PC
Flujo Normal: 1-UD_HOST manda a US flujo para subir a BBDD HOST. 2-US inserta flujo en la BBDD.

Caso de uso: Solicita Ejecución Flujo Producción
Actores: Usuario Diseñador Host, Usuario Sistemas
Tipo
Descripción
Requerimientos
Curso normal de los sucesos
Precondición: Petición de subida Flujo al Host
Flujo Normal: 1-UD_HOST solicita la ejecución en producción del último flujo a US.
Flujo Alternativo: 1-UD_HOST solicita la ejecución en producción de cualquier otro flujo con estado activo a US.

5. Evaluación de Tecnologías

En este apartado se van evaluar los lenguajes de programación que son más óptimos para la implementación de este producto, así como los gestores de BBDD más acordes al lenguaje de programación como también al entorno en el que tienen que ejecutarse.

5.1 Lenguajes de Programación

A la hora de tomar la decisión del lenguaje a emplear para la construcción de este motor, se barajaron las dos opciones posibles para la ejecución en entornos HOST. Estas 2 opciones fueron: COBOL y Java.

5.1.1 COBOL

El lenguaje COBOL (acrónimo de COmmon Business Oriented Language, Lenguaje Común Orientado a Negocios) fue creado en el año 1960 con el objetivo de crear un lenguaje de programación universal que pudiera ser usado en cualquier ordenador, ya que en los años 1960 existían numerosos modelos de ordenadores incompatibles entre sí, y que estuviera orientado principalmente a los negocios, es decir, a la llamada informática de gestión.

Historia

En la creación de este lenguaje participó la comisión CODASYL, compuesta por fabricantes de ordenadores, usuarios y el Departamento de Defensa de Estados Unidos en mayo de 1959. La definición del lenguaje se completó en poco más de seis meses, siendo aprobada por la comisión en enero de 1960. El lenguaje COBOL fue diseñado inspirándose en el lenguaje Flow-Matic de Grace Hopper y el IBM COMTRAN de Bob Bemer, ya que ambos formaron parte de la comisión.

Gracias a la ayuda de los usuarios COBOL evolucionó rápidamente y fue revisado de 1961 a 1965 para añadirle nuevas funcionalidades. En 1968 salió la primera versión ANSI del lenguaje, siendo revisada posteriormente en 1974 (COBOL ANS-74), 1985 (COBOL ANS-85, ampliado en 1989 con funciones matemáticas, finalizando el estándar actual más usado, conocido como COBOL-ANSI), y en 2002 (COBOL ANS-2002). Desde el año 2007 se viene preparando una nueva revisión del lenguaje.

Además, existe una versión conocida como COBOL ENTERPRISE, actualizada regularmente y lanzada en 1991, usada generalmente en sistemas Host.

Características

- COBOL fue dotado de unas excelentes capacidades de autodocumentación.
- Una buena gestión de archivos y una excelente gestión de los tipos de datos para la época, a través de la conocida sentencia PICTURE para la definición de campos estructurados.
- Para evitar errores de redondeo en los cálculos que se producen al convertir los números a binario y que son inaceptables en temas comerciales, COBOL puede emplear y emplea por defecto números en base diez.
- Para facilitar la creación de programas en COBOL, la sintaxis del mismo fue creada de forma que fuese parecida al idioma inglés, evitando el uso de símbolos que se impusieron en lenguajes de programación posteriores.

Pese a esto, a comienzos de los ochenta se fue quedando anticuado respecto a los nuevos paradigmas de programación y a los lenguajes que los implementaban. En la revisión de 1985 se solucionó, incorporando a COBOL variables locales, recursividad, reserva de memoria dinámica y programación estructurada.

En la revisión de 2002 se le añadió orientación a objetos, aunque desde la revisión de 1974 se podía crear un entorno de trabajo similar a la orientación a objetos, y un método de generación de pantallas gráficas estandarizado.

Antes de la inclusión de las nuevas características en el estándar oficial, muchos fabricantes de compiladores las añadían de forma no estándar. En la actualidad este proceso se está viendo con la integración de COBOL con Internet. Existen varios compiladores que permiten emplear COBOL como lenguaje de scripting y de servicio web. También existen compiladores que permiten generar código COBOL para la plataforma .NET y EJB.

Empleo

Pese a que muchas personas creen que el lenguaje COBOL está en desuso, la realidad es que casi todos los sistemas que requieren gran capacidad de procesamiento por lotes (Batch), tanto las entidades bancarias como otras grandes empresas con sistemas mainframes utilizan COBOL. Esto permite garantizar la compatibilidad de los sistemas antiguos con los más modernos, así como tener la seguridad de que el lenguaje es perfectamente estable y probado. Según un informe de Gartner Group de 2005, el 75% de los datos generados por

negocios son procesados por programas creados en COBOL, y en otro informe de 1997 estima que el 80% de los 300.000 millones de líneas de código existentes están creados en COBOL, escribiéndose 5.000 millones de líneas nuevas de COBOL cada año. Con todo eso, hoy por hoy, la programación en COBOL es uno de los negocios más rentables del mundo de la informática. En el resto de aplicaciones el COBOL ha caído en desuso, remplazado por lenguajes más modernos o versátiles.

5.1.2 JAVA

Java es un lenguaje de programación orientado a objetos, desarrollado por Sun Microsystems a principios de los años 90. El lenguaje en sí mismo toma mucha de su sintaxis de C y C++, pero tiene un modelo de objetos más simple y elimina herramientas de bajo nivel, que suelen inducir a muchos errores, como la manipulación directa de punteros o memoria.

Las aplicaciones Java están típicamente compiladas en un *bytecode*, aunque la compilación en código máquina nativo también es posible. En el tiempo de ejecución, el *bytecode* es normalmente interpretado o compilado a código nativo para la ejecución, aunque la ejecución directa por hardware del *bytecode* por un procesador Java también es posible.

La implementación original y de referencia del compilador, la máquina virtual y las bibliotecas de clases de Java fueron desarrolladas por Sun Microsystems en 1995. Desde entonces, Sun ha controlado las especificaciones, el desarrollo y evolución del lenguaje a través del Java Community Process, si bien otros han desarrollado también implementaciones alternativas de estas tecnologías de Sun, algunas incluso bajo licencias de software libre.

Entre diciembre de 2006 y mayo de 2007, Sun Microsystems liberó la mayor parte de sus tecnologías Java bajo la licencia GNU GPL, de acuerdo con las especificaciones del Java Community Process, de tal forma que prácticamente todo el Java de Sun es ahora software libre (aunque la biblioteca de clases de Sun que se requiere para ejecutar los programas Java aún no lo es).

Historia

Java se creó como una herramienta de programación para ser usada en un proyecto de set-top-box en una pequeña operación denominada *the Green Project* en Sun Microsystems en el año 1991. El equipo (*Green Team*), compuesto por trece personas y dirigido por James Gosling, trabajó durante 18 meses en Sand Hill Road en Menlo Park en su desarrollo.

El lenguaje se denominó inicialmente *Oak* (por un roble que había fuera de la oficina de Gosling), luego pasó a denominarse *Green* tras descubrir que *Oak* era ya una marca comercial registrada para adaptadores de tarjetas gráficas y finalmente se renombró a *Java*.

El término Java fue acuñado en una cafetería frecuentada por algunos de los miembros del equipo. Pero no está claro si es un acrónimo o no, aunque algunas fuentes señalan que podría tratarse de las iniciales de sus creadores: **J**ames Gosling, **A**rthur **V**an Hoff, y **A**ndy Bechtolsheim. Otros abogan por el siguiente acrónimo, **J**ust **A**nother **V**ague **A**cronym ("sólo otro acrónimo ambiguo más"). La hipótesis que más fuerza tiene es la que Java debe su nombre a un tipo de café disponible en la cafetería cercana, de ahí que el icono de java sea una taza de café caliente. Un pequeño signo que da fuerza a esta teoría es que los 4 primeros bytes (el *número mágico*) de los archivos .class que genera el compilador, son en hexadecimal, 0xCAFEBABE. A pesar de todas estas teorías, el nombre fue sacado al parecer de una lista aleatoria de palabras.

Los objetivos de Gosling eran implementar una máquina virtual y un lenguaje con una estructura y sintaxis similar a C++. Entre junio y julio de 1994, tras una sesión maratoniana de tres días entre John Gaga, James Gosling, Joy Naughton, Wayne Rosing y Eric Schmidt, el equipo reorientó la plataforma hacia la Web. Sintieron que la llegada del navegador web Mosaic, propiciaría que Internet se convirtiese en un medio interactivo, como el que pensaban era la televisión por cable. Naughton creó entonces un prototipo de navegador, WebRunner, que más tarde sería conocido como HotJava.

En 1994, se les hizo una demostración de HotJava y la plataforma Java a los ejecutivos de Sun. Java 1.0a pudo descargarse por primera vez en 1994, pero hubo que esperar al 23 de mayo de 1995, durante las conferencias de SunWorld, a que vieran la luz pública Java y HotJava, el navegador Web. El acontecimiento fue anunciado por John Gage, el Director Científico de Sun Microsystems. El acto estuvo acompañado por una pequeña sorpresa adicional, el anuncio por parte de Marc Andreessen, Vicepresidente Ejecutivo de Netscape, de que Java sería soportado en sus navegadores. El 9 de enero del año siguiente, 1996, Sun fundó el grupo empresarial JavaSoft para que se encargase del desarrollo tecnológico. Dos semanas más tarde la primera versión de Java fue publicada.

La promesa inicial de Gosling era *Write Once, Run Anywhere* (Escríbelo una vez, ejecútalo en cualquier lugar), proporcionando un lenguaje independiente de la plataforma y un entorno de ejecución (la JVM) ligero y gratuito para las plataformas más populares de forma que los binarios (bytecode) de las aplicaciones Java pudiesen ejecutarse en cualquier plataforma.

El entorno de ejecución era relativamente seguro y los principales navegadores web pronto incorporaron la posibilidad de ejecutar applets Java incrustadas en las páginas web.

Java ha experimentado numerosos cambios desde la versión primigenia, JDK 1.0, así como un enorme incremento en el número de clases y paquetes que componen la biblioteca estándar.

Desde J2SE 1.4, la evolución del lenguaje ha sido regulada por el JCP (Java Community Process), que usa *Java Specification Requests* (JSRs) para proponer y especificar cambios en la plataforma Java. El lenguaje en sí mismo está especificado en la *Java Language Specification* (JLS), o Especificación del Lenguaje Java. Los cambios en los JLS son gestionados en JSR 901.

Además de los cambios en el lenguaje, con el paso de los años se han efectuado muchos más cambios dramáticos en la biblioteca de clases de Java (*Java class library*) que ha crecido de unos pocos cientos de clases en JDK 1.0 hasta más de tres mil en J2SE 5.0. APIs completamente nuevas, como Swing y Java2D, han sido introducidas y muchos de los métodos y clases originales de JDK 1.0 están obsoletos.

En el 2005 se calcula en 4,5 millones el número de desarrolladores y 2.500 millones de dispositivos habilitados con tecnología Java.

Características.

El lenguaje Java se creó con cinco objetivos principales:

1. Debería usar la metodología de la programación orientada a objetos.
2. Debería permitir la ejecución de un mismo programa en múltiples sistemas operativos.
3. Debería incluir por defecto soporte para trabajo en red.
4. Debería diseñarse para ejecutar código en sistemas remotos de forma segura.
5. Debería ser fácil de usar y tomar lo mejor de otros lenguajes orientados a objetos, como C++.

Para conseguir la ejecución de código remoto y el soporte de red, los programadores de Java a veces recurren a extensiones como CORBA (Common Object Request Broker Architecture), Internet Communications Engine o OSGi respectivamente.

Orientado a objetos

La primera característica, orientado a objetos ("OO"), se refiere a un método de programación y al diseño del lenguaje. Aunque hay muchas interpretaciones para OO, una primera idea es diseñar el software de forma que los distintos tipos de datos que usen estén unidos a sus operaciones. Así, los datos y el código (funciones o métodos) se combinan en entidades llamadas objetos. Un objeto puede verse como un paquete que contiene el "comportamiento" (el código) y el "estado" (datos). El principio es separar aquello que cambia de las cosas que permanecen inalterables. Frecuentemente, cambiar una estructura de datos implica un cambio en el código que opera sobre los mismos, o viceversa. Esta separación en objetos coherentes e independientes ofrece una base más estable para el diseño de un sistema software. El objetivo es hacer que grandes proyectos sean fáciles de gestionar y manejar, mejorando como consecuencia su calidad y reduciendo el número de proyectos fallidos. Otra de las grandes promesas de la programación orientada a objetos es la creación de entidades más genéricas (objetos) que permitan la reutilización del software entre proyectos, una de las premisas fundamentales de la Ingeniería del Software. Un objeto genérico "cliente", por ejemplo, debería en teoría tener el mismo conjunto de comportamiento en diferentes proyectos, sobre todo cuando estos coinciden en cierta medida, algo que suele suceder en las grandes organizaciones. En este sentido, los objetos podrían verse como piezas reutilizables que pueden emplearse en múltiples proyectos distintos, posibilitando así a la industria del software a construir proyectos de envergadura empleando componentes ya existentes y de comprobada calidad; conduciendo esto finalmente a una reducción drástica del tiempo de desarrollo. Podemos usar como ejemplo de objeto el aluminio. Una vez definidos datos (peso, maleabilidad, etc.), y su "comportamiento" (soldar dos piezas, etc.), el objeto "aluminio" puede ser reutilizado en el campo de la construcción, del automóvil, de la aviación, etc.

La reutilización del software ha experimentado resultados dispares, encontrando dos dificultades principales: el diseño de objetos realmente genéricos es pobremente comprendido, y falta una metodología para la amplia comunicación de oportunidades de reutilización. Algunas comunidades de "código abierto" (open source) quieren ayudar en este problema dando medios a los desarrolladores para diseminar la información sobre el uso y versatilidad de objetos reutilizables y bibliotecas de objetos.

Independencia de la plataforma

La segunda característica, la independencia de la plataforma, significa que programas escritos en el lenguaje Java pueden ejecutarse igualmente en cualquier tipo de hardware. Este es el significado de ser capaz de escribir un programa una vez y que pueda ejecutarse en cualquier dispositivo, tal como reza el axioma de Java, '"write once, run anywhere"'.

Para ello, se compila el código fuente escrito en lenguaje Java, para generar un código conocido como "bytecode" (específicamente Java bytecode)—instrucciones máquina simplificadas específicas de la plataforma Java. Esta pieza está "a medio camino" entre el código fuente y el código máquina que entiende el dispositivo destino. El bytecode es ejecutado entonces en la máquina virtual (JVM), un programa escrito en código nativo de la plataforma destino (que es el

que entiende su hardware), que interpreta y ejecuta el código. Además, se suministran bibliotecas adicionales para acceder a las características de cada dispositivo (como los gráficos, ejecución mediante hebras o threads, la interfaz de red) de forma unificada. Se debe tener presente que, aunque hay una etapa explícita de compilación, el bytecode generado es interpretado o convertido a instrucciones máquina del código nativo por el compilador JIT (Just In Time).

Hay implementaciones del compilador de Java que convierten el código fuente directamente en código objeto nativo, como GCJ. Esto elimina la etapa intermedia donde se genera el bytecode, pero la salida de este tipo de compiladores sólo puede ejecutarse en un tipo de arquitectura.

La licencia sobre Java de Sun insiste que todas las implementaciones sean "compatibles". Esto dio lugar a una disputa legal entre Microsoft y Sun, cuando éste último alegó que la implementación de Microsoft no daba soporte a las interfaces RMI y JNI además de haber añadido características ''dependientes'' de su plataforma. Sun demandó a Microsoft y ganó por daños y perjuicios (unos 20 millones de dólares) así como una orden judicial forzando la acatación de la licencia de Sun. Como respuesta, Microsoft no ofrece Java con su versión de sistema operativo, y en recientes versiones de Windows, su navegador Internet Explorer no admite la ejecución de applets sin un conector (o plugin) aparte. Sin embargo, Sun y otras fuentes ofrecen versiones gratuitas para distintas versiones de Windows.

Las primeras implementaciones del lenguaje usaban una máquina virtual interpretada para conseguir la portabilidad. Sin embargo, el resultado eran programas que se ejecutaban comparativamente más lentos que aquellos escritos en C o C++. Esto hizo que Java se ganase una reputación de lento en rendimiento. Las implementaciones recientes de la JVM dan lugar a programas que se ejecutan considerablemente más rápido que las versiones antiguas, empleando diversas técnicas, aunque sigue siendo mucho más lento que otros lenguajes.

La primera de estas técnicas es simplemente compilar directamente en código nativo como hacen los compiladores tradicionales, eliminando la etapa del bytecode. Esto da lugar a un gran rendimiento en la ejecución, pero tapa el camino a la portabilidad. Otra técnica, conocida como compilación JIT (Just In Time, o '''compilación al vuelo'''), convierte el bytecode a código nativo cuando se ejecuta la aplicación. Otras máquinas virtuales más sofisticadas usan una '''recompilación dinámica''' en la que la VM es capaz de analizar el comportamiento del programa en ejecución y recompila y optimiza las partes críticas. La recompilación dinámica puede lograr mayor grado de optimización que la compilación tradicional (o estática), ya que puede basar su trabajo en el conocimiento que de primera mano tiene sobre el entorno de ejecución y el conjunto de clases cargadas en memoria. La compilación JIT y la recompilación dinámica permiten a los programas Java aprovechar la velocidad de ejecución del código nativo sin por ello perder la ventaja de la portabilidad en ambos.

La portabilidad es técnicamente difícil de lograr, y el éxito de Java en ese campo ha sido dispar. Aunque es de hecho posible escribir programas para la plataforma Java que actúen de forma correcta en múltiples plataformas de distinta arquitectura, el gran número de estas con pequeños errores o inconsistencias

llevan a que a veces se parodie el eslogan de Sun, "Write once, run anywhere" como "Write once, debug everywhere" (o "Escríbelo una vez, ejecútalo en cualquier parte" por "Escríbelo una vez, depúralo en todas partes")

El concepto de independencia de la plataforma de Java cuenta, sin embargo, con un gran éxito en las aplicaciones en el entorno del servidor, como los Servicios Web, los Servlets, los Java Beans, así como en sistemas empotrados basados en OSGi, usando entornos Java empotrados.

El recolector de basura

En Java el problema de las fugas de memoria se evita en gran medida gracias a la recolección de basura (o *automatic garbage collector*). El programador determina cuándo se crean los objetos y el entorno en tiempo de ejecución de Java (Java runtime) es el responsable de gestionar el ciclo de vida de los objetos. El programa, u otros objetos pueden tener localizado un objeto mediante una referencia a éste. Cuando no quedan referencias a un objeto, el recolector de basura de Java borra el objeto, liberando así la memoria que ocupaba previniendo posibles fugas (ejemplo: un objeto creado y únicamente usado dentro de un método sólo tiene entidad dentro de éste; al salir del método el objeto es eliminado). Aun así, es posible que se produzcan fugas de memoria si el código almacena referencias a objetos que ya no son necesarios—es decir, pueden aún ocurrir, pero en un nivel conceptual superior. En definitiva, el recolector de basura de Java permite una fácil creación y eliminación de objetos, mayor seguridad y puede que más rápida que en C++

5.2 Toma de decisión Lenguaje Programación

En un primer lugar, la opción que cobraba más fuerza fue la de utilizar la herramienta de desarrollo Java para llevar a cabo la implementación, pero existieron dos razones por las que se desecharon. La primera y más importante es que no todos los HOST soportan aplicaciones implementadas en Java (solamente las versiones más recientes). El segundo punto que más me preocupaba era el tema de la eficiencia, ya que por la experiencia que he tenido, las aplicaciones en lenguaje Java pueden llegar a ser más lentas que las escritas en otros lenguajes (este punto va justamente ligado con la teóricamente versatilidad de ser compatible con cualquier sistema). Por eso, finalmente opté por realizar la programación en el lenguaje COBOL, que a pesar de tratarse de un lenguaje que lleva muchos años usándose, y hay gente que lo considera obsoleto o en desuso, siguen siendo mayoritarias las implantaciones de programas en este lenguaje en el ámbito empresarial. No obstante, y a pesar de que se haya realizado la implementación en este lenguaje, se han utilizado las opciones del lenguaje más avanzadas que posee, para así poder afinar el rendimiento de la aplicación.

Entre algunas de estas funciones, que en muchos casos son desconocidas en este lenguaje, han sido: el uso de punteros, estructuras dinámicas, métodos de debug, variables con coma flotante, etc.

5.3 Sistemas Gestores de Bases de datos

Al plantear el sistema de BBDD a usar en la parte del aplicativo HOST, los principales nombres de sistemas de BBDD que son más empleados en los HOST son o bien el más que conocido DB2, o bien el sistema ORACLE cada vez en más uso.

A continuación vamos a ver las características de cada uno de ellas.

5.3.1 ORACLE

La BD Oracle pertenece al grupo Oracle Corporation, y la última versión disponible estable existente en el mercado es la versión 11g.

Está BBDD es multiplataforma y la licencia es de uso privado.

Oracle es básicamente una herramienta cliente/servidor, un sistema de gestión de base de datos relacional (o RDBMS por el acrónimo en inglés de Relational Data Base Management System), desarrollado por Oracle Corporation.

Una base de datos Oracle está almacenada físicamente en ficheros, y la correspondencia entre los ficheros y las tablas es posible gracias a las estructuras internas de las bases de datos, que permiten que diferentes tipos de datos estén almacenados físicamente de forma separada. Está división lógica se hace gracias a los tablespaces (espacios de tablas).

Las características principales son:

- Posee una herramienta de administración gráfica que es muy intuitiva de utilizar.
- Ayuda a analizar datos y a efectuar recomendaciones concernientes para mejorar el rendimiento y la eficiencia en el manejo de aquellos datos que se encuentran almacenados.
- Apoya en el diseño y optimización de modelos de datos.
- Tiene asistentes para los desarrolladores que tienen conocimientos de SQL y de construcción de procedimientos almacenados y triggers, entre otros.
- Posee igual interacción en todas las plataformas (Windows, Unix, Macintosh y Mainframes). Esto es debido a que más del 80% del código interno de Oracle es igual a los establecidos en todas las plataformas de Sistemas Operativos.
- Soporta bases de datos de todos los tamaños, desde pequeñas a tamaños de gigabytes.
- Garantiza de forma muy optima la seguridad de acceso y la de los mismos datos.
- En la actualidad es la primera Base de Datos usada en el mundo empresarial.

5.3.2 DB2

La Base de Datos DB2 pertenece a IBM (International Business Machines). La última versión disponible es la 9.7.

Esta BBDD es multiplataforma y la licencia de uso es privada.

DB2 viene a ser el producto principal de IBM como estrategia de Data Management. Es un sistema para la administración de bases de datos realacionales (RDBMS) multiplataforma, especialmente diseñada para ambientes distribuidos, permitiendo que los usuarios locales compartan información con los recursos centrales.

Las principales características son:

- Permite agilizar el tiempo de respuesta de las consultas
- Recuperación de datos utilizando accesos de solo índices.
- Predicados correlacionados
- Tablas de resumen y Uniones hash
- La seguridad se garantiza en combinación de seguridad externa y control interno de acceso para protección de los datos.
- DB2 protege los datos para evitar pérdidas, contra el acceso desautorizado o entradas inválidas.
- La administración de la BBDD se puede realizar desde cualquier terminal.
- Excelente soporte técnico y uso principal extendido en máquinas IBM.
- DB2 se basa en dos ejes que lo hacen muy eficaz en rendimiento: utiliza un sistema multiprocesador (SMP) simétrico y un sistema de procesador paralelo masivo.

5.3.3 Toma de decisión BBDD.

Es realmente difícil poder decidir entre los dos gestores más usados en el mundo empresarial, por una parte ORACLE que es el más usado y es el que está teniendo mayor crecimiento en los últimos años, incluso llegando a adquirir a la tan conocida INFORMIX. Por otra parte esta DB2 que es un gestor de base de datos que es el predominante en los Mainframes.

La razón que más ha cobrado peso ha sido precisamente su extensión en los Mainframes, ya que junto con COBOL sigue siendo a día de hoy lo más usado en estos sistemas. En buena parte la culpa de ello es que los sistemas Mainframe suelen ser IBM y por ello se confió más en el uso de las BD DB2. Aunque poco a poco se está empezando a extender el uso de ORACLE y JAVA propiedad también de Oracle Corporation.

Por todo lo anteriormente citado, y por existir mayor compatibilidad de librerías para su compilación y procedimientos entre COBOL y DB2 gana esta y es la que se ha usado para poder realizar los procedimientos HOST.

6. Diseño

Para realizar la implementación del motor hemos hecho servir lo que llamaremos "como tiras polacas" que no son más que una codificación propia de instrucciones, usando como base la notación polaca inversa (como las que usan las calculadoras científicas HP). Estas instrucciones(tiras polacas), han sido diseñadas para poder soportar todas las funciones que hemos requerido implementar y así poder llegar a gestionar flujos de toma de decisión complejos (no-lineales).

A continuación haremos una breve mención al funcionamiento de la notación polaca inversa. Más adelante cuando hagamos referencia al fichero que contiene la codificación de instrucciones entraremos más en detalle y explicaremos la codificación empleada que hemos usado partiendo de la notación polaca inversa.

Su principio es el de evaluar los datos directamente cuando se introducen y manejarlos dentro de una estructura LIFO (Last In First Out), lo que optimiza los procesos a la hora de programar.

Básicamente la diferencias con el método algebraico o notación de infijo es que, al evaluar los datos directamente al introducirlos, no es necesario ordenar la evaluación de los mismos, y que para ejecutar un comando, primero se deben introducir todos sus argumentos. Así, para hacer una suma 'a+b=c' el RPN lo manejaría a b +, dejando el resultado 'c' directamente.

Nótese que la notación polaca inversa no es literalmente la imagen especular de la notación polaca: el orden de los operandos es igual en las tres notaciones (infijo, prefijo o polaca, y postfijo o polaca inversa). Lo que cambia es el lugar donde va el operador. En la notación infija, el operador va en el medio de los operandos, mientras que en la notación polaca va antes y en la notación polaca inversa va después. Así pues, "640 / 16" (en notación de infijo), se escribe como "/ 640 16" (en notación polaca) y como "640 16 /" en notación polaca inversa. El orden de los operandos es importante cuando se manejan operadores no conmutativos (como la resta o la división), así, si dividimos 10 entre 2, por ejemplo, en las tres notaciones se debe escribir de la siguiente manera: "10 / 2", "/ 10 2", "10 2 /".

6.1 Definición de variables y tipos de datos del motor

Normalmente en la programación de flujos o modelos para banca, una de las puntos básicos que se exigen a la hora de programar o realizar nuevos desarrollos consiste en que las interfaces de entrada y de salida vengan prefijados. En pocas palabras te los facilitan desde los equipos de diseño y desarrollo, de forma que ni tan siquiera se tienen que fijar los tipos de datos de las variables a la hora de realizar un nuevo programa. Esto normalmente se suele realizar de esta forma, ya que en muchas ocasiones los nuevos programas que se realizan no son más que restimaciones o lo que es lo mismo modificaciones simples de la lógica del programa, cambio de pesos, políticas, modificación de las variables que puntúan en el modelo, etc. Otro motivo, por el cual dar una estructura fija, puede ser por la complejidad de obtener la información. Tampoco es factible en otros casos por temas de costes o tiempos ya que implica la realización de análisis adhoc de la información que se requiere para la realización del programa. Por ello, se pasan estructuras de datos que con mucha frecuencia contienen más información de la necesaria, es decir con variables de las cuales no vamos a hacer uso.

En cualquier programa a fin de cuentas, se suele tener una interfaz de entrada y otra de salida, para los cuales se nos proporcionan datos para que sean evaluados o calculados y luego se obtienen unos resultados que es lo que finalmente se devuelve.

En los puntos que vienen a continuación se pretende explicar unas estructuras de datos, las cuales nos van a permitir obtener una gran flexibilidad, ya que nos permitirán adaptarnos a los requerimientos de estructuras de datos fijas que se suelen emplear en la industria, pero sin perder funcionalidad a la hora de aplicarlo a los flujos. Adicionalmente nos va a permitir obtener una gran independencia de los datos prefijados por el cliente, ya que vamos a despreciar en gran medida las definiciones de datos proporcionadas, especialmente con las variables de salida, por lo que vamos a obtener unas ejecuciones de información que contengan además gran fiabilidad en los datos (ya que no perderemos precisión), cosa que no ocurre en los análisis que hemos realizado de la competencia.

6.1.1 Estructura de definición de variables

Esta estructura nos va a permitir definir únicamente las variables que van a ser empleadas en el motor de flujos. En este módulo no se tienen en cuenta las variables innecesarias. Está dividida en tres subestructuras o áreas y cada una de ellas contiene la definición de las tres posibles variables que se han contemplado en la implementación: "Variables de entrada", "Variables de

Cálculo” y “Variables de Salida”.

En la subestructura de variables de entrada, se incluirán todas aquellas variables de las cuales en algún punto de la ejecución del flujo, requiramos su lectura y se nos haya pasado por parámetro. Sobre estas variables no podremos modificar o alterar la información que contengan, por lo que si se intenta el proceso de ejecución retornará un error.

Las variables de cálculo, nos permitirán realizar cualquier tipología de cálculo, sin requerir que la variable a la cual se le asigne el resultado haya sido declarada en la estructura de salida. Sobre este tipo de variables se puede realizar cualquier tipo de operación, ya sean de lectura o escritura.

Por último las variables de salida, también nos permiten disponer de las mismas funcionalidades que sobre las variables de cálculo, no obstante, al tener posiciones fijas, sobre ellas se pueden producir desbordamientos u overflows. Adicionalmente, cabe decir, que cualquier dato que se haya asignado a este tipo de variables se retornará por salida.

A continuación pasamos a explicar en detalle la estructura.

La estructura de definición de los formatos de variables consta realmente de 4 partes:

- En la primera vienen unos números que nos indicarán el número de elementos de los que constarán las otras 3 partes. Es decir el primer número nos indicará el número de variables de entrada a leer, el segundo número nos indicará el número de variables de cálculo o internas que usa el motor y el tercer número nos indica el número de variables que tenemos definidas como de salida.

Una peculiaridad adicional a destacar, consiste en que nosotros podemos tener definidas muchas tipologías de variables para realizar cálculos o simplemente de entrada o de salida, pero no necesariamente las tenemos que usar, con ello queremos decir, y como se vera más adelante, que la herramienta permite personalizar qué variables de entrada o salida queremos coger o devolver.

- La segunda parte consiste en la definición de los tipos de datos de las variables de entrada. A continuación explicamos en detalle cada uno de los caracteres que componen cada definición de variable:

4 dígitos que nos indican el número de variable

1 carácter que nos indicará si la variable es numérica o alfanumérica. Sus valores serán (N o A).

2 caracteres que nos indicarán, en el caso de si la variable es alfanumérica, la longitud total de caracteres de la variable. En caso de ser numérica, nos indicará también la longitud total de la variable.

2 caracteres que nos indicarán, en el caso de ser la variable numérica, cuántas

posiciones de la variable numérica forman la parte decimal.

16 caracteres para indicar el nombre lógico de la variable

2 caracteres para identificar si la variable es una variable múltiple o única. Con variable única queremos decir que en el fichero o estructura que contenga los datos sólo se deba leer una vez por operación. Con variable múltiple queremos decir que se va a leer la variable N veces. Un ejemplo de variable única en un programa de banca podría ser la variable "Plazo de la operación" o "la cuota asociada a la operación". En cambio, un ejemplo de variables múltiples podría ser la variable "Edad" o "Estado civil", "Ingresos" ya que normalmente en una operación de préstamos al consumo y sobretodo hipotecarias puede ir más de un interviniente en la misma operación para solicitar el préstamo, por tanto podemos tener diferentes ingresos o estados civiles o edades en función de cada uno de los titulares, en cambio cosas comunes de la operación serían el plazo o la cuota que tendríamos que pagar que serían los mismos por muchos titulares existentes en la operación.

Esta variable toma por valores; UN (Única) y M1 (Múltiple).

- El tercer bloque, consiste en la definición de las variables de cálculo que son necesarias para llevar a cabo la programación del modelo y que no forman parte de las interfaces ni de entrada ni de salida. El formato es muy similar al que hemos visto con la entrada.

4 dígitos que nos indican el número de variable

1 carácter que nos indicará si la variable es numérica o alfanumérica. Sus valores serán (N o A).

2 caracteres que nos indicarán en el caso de si la variable es alfanumérica la longitud total de caracteres de la variable. En caso de ser numérica, nos indicará la longitud total de la variable

2 caracteres que nos indicarán en el caso de ser la variable numérica cuántas posiciones de la variable numérica forman parte de la parte decimal.

- El cuarto y último bloque son las variables de salida, cuyo formato es el mismo que para las variables de entrada:

4 dígitos que nos indican el número de variable o nombre físico

1 carácter que nos indicará si la variable es numérica o alfanumérica. Sus valores serán (N o A).

2 caracteres que nos indicarán en el caso de si la variable es alfanumérica la longitud total de caracteres de la variable. En caso de ser numérica, también nos indicará la longitud total de la variable

2 caracteres que nos indicarán en el caso de ser la variable numérica cuántas posiciones de la variable numérica forman parte de la parte decimal.

16 caracteres para indicar el nombre lógico de la variable

2 caracteres para identificar si la variable es una variable múltiple o única. Esta variable toma por valores; UN (Única) y M1 (Múltiple).

A continuación se muestra la copy COBOL que lee esta estructura de datos (GVARS):

```
01 :TAG:REG-VARIABLES.
      03 :TAG:NUM-VARS-ORIG            PIC 9(04).
      03 :TAG:NUM-VARS-ARTIF           PIC 9(04).
      03 :TAG:NUM-VARS-SALIDA          PIC 9(04).
      03 :TAG:VARS-ORIGINALES
        OCCURS 1 to :TAGVI: DEPENDING ON
        :TAG:NUM-VARS-ORIG
        INDEXED BY :TAG:IN-VARS-ORIGINALES.
        10 :TAG:VO-NLOGICO             PIC 9(04).
        10 :TAG:VO-TIPO-NA             PIC X(01).
        10 :TAG:VO-LPTOT               PIC 9(02).
        10 :TAG:VO-LPDEC               PIC 9(02).
        10 :TAG:VO-NFISICO             PIC X(16).
        10 :TAG:VO-TIPO-MU             PIC X(02).
      03 :TAG:VARS-ARTIFICIALES
        OCCURS 1 TO :TAGVA: DEPENDING ON
        :TAG:NUM-VARS-ARTIF
        INDEXED BY :TAG:IN-VARS-ARTIFICIALES.
        10 :TAG:VA-NLOGICO             PIC 9(04).
        10 :TAG:VA-TIPO-NA             PIC X(01).
        10 :TAG:VA-LPTOT               PIC 9(02).
        10 :TAG:VA-LPDEC               PIC 9(02).
       10 :TAG:VA-NFISICO              PIC X(16).
      03 :TAG:VARS-SALIDA
        OCCURS 1 TO :TAGVO: DEPENDING ON
        :TAG:NUM-VARS-SALIDA
        INDEXED BY :TAG:IN-VARS-SALIDA.
        10 :TAG:VS-NLOGICO             PIC 9(04).
        10 :TAG:VS-TIPO-NA             PIC X(01).
        10 :TAG:VS-LPTOT               PIC 9(02).
        10 :TAG:VS-LPDEC               PIC 9(02).
        10 :TAG:VS-NFISICO             PIC X(16).
        10 :TAG:VS-TIPO-MU             PIC X(02).
```

6.1.2 Definición de la estructura cabeceras de entrada

Como se ve en la copy o estructura de definición donde explicamos el tipo de datos de las variables (6.1.1), no se indica en que posición debemos recoger los datos para las variables de entrada, ni tampoco en que posición debemos de dejarlos para las variables de salida. Para definir en que posición se encuentran las variables dentro de las interfaces, estructuras de comunicación o en la ristra de caracteres que se pasen por parámetro, se ha definido la estructura CABECERA_IN. A continuación pasaremos a definir cada uno de las partes que componen el fichero y lo que se encuentra en cada una de las partes.

En la primera parte de la estructura nos encontramos un campo numérico que nos indica cuantas variables vamos a tener detalladas en la interface de CABECERA_IN. Este campo podrá tomar como máximo el valor 9999 (9999 variables de entrada)

En la segunda parte de la estructura, nos encontramos con la definición de la posición que ocupa cada una de las variables dentro de la interface de datos de entrada. A continuación detallamos su composición:

16 primeros caracteres nos indican el nombre de la variable.

5 caracteres nos indican la posición inicial que ocupará esta variable dentro del fichero de datos.

5 caracteres nos indicarán la posición final que ocupará esta variable dentro del fichero de datos.

2 caracteres adicionales para indicarnos si la variable forma parte de las variables únicas o múltiples.

A continuación se muestra la copy que lee esta estructura de datos (GCABEC):

```
 01 :TAG:REG-CABECERA-:TAG1:.
03 :TAG:NUM-REGS-CABECERA-:TAG1:       PIC 9(04).
03 :TAG:PARTE-MULTIPLE-:TAG1:          PIC 9(04).
03 :TAG:REGS-CABECERA-:TAG1:
   OCCURS :TAGV: TIMES.
  05 :TAG:NOM-REG-CABECERA-:TAG1:     PIC X(16).
  05 :TAG:INICIO-REGS-CABECERA-:TAG1: PIC 9(04).
  05 :TAG:FINAL-REGS-CABECERA-:TAG1:  PIC 9(04).
  05 :TAG:PARTE-TOTAL-:TAG1:          PIC 9(02).
  05 :TAG:PARTE-DECIMAL-:TAG1:        PIC 9(02).
  05 :TAG:NOMBRE-LOGICO-:TAG1:        PIC 9(04).
  05 :TAG:TIPO-NA-:TAG1:              PIC X(01).
  05 :TAG:TIPO-MU-:TAG1:              PIC X(02).
   05   TAG:ENCONTRADO-:TAG1:          PIC X(01).
```

6.1.3 Definición de la estructura cabeceras de salida

Al igual que existe una estructura para definir dónde se encuentran cada una de las variables de la interface de datos de entrada, existe otra estructura con el mismo esquema que sirve para indicar al motor en que posiciones tiene que dejar el resultado de cada una de las variables que se quieren que se muestren en la salida.

En la primera parte nos encontramos un campo numérico que nos indicará cuántas variables vamos a tener detalladas en la estructura de CABECERA_OUT. Este campo podrá tomar como máximo el valor 9999 (9999 variables de salida)

En la segunda parte de la estructura nos encontramos con la definición de las

posiciones que ocupan cada una de las variables dentro de la estructura de datos de salida. A continuación detallamos su composición:

16 primeros caracteres nos indican el nombre de la variable.

5 caracteres nos indican la posición inicial que ocupará esta variable dentro del fichero de datos.

5 caracteres nos indicarán la posición final que ocupará esta variable dentro del fichero de datos.

2 caracteres adicionales para indicarnos si la variable forma parte de las variables únicas o múltiples.

Como se puede apreciar la estructura de datos es idénticamente la misma que para la estructura de CABECERA_IN, incluso el fichero de la COPY que permite realizar la lectura de la estructura de cabeceras de salida es el mismo (GCABEC):

```
01 :TAG:REG-CABECERA-:TAG1:.
03 :TAG:NUM-REGS-CABECERA-:TAG1:       PIC 9(04).
03 :TAG:PARTE-MULTIPLE-:TAG1:          PIC 9(04).
03 :TAG:REGS-CABECERA-:TAG1:
   OCCURS :TAGV: TIMES.
  05 :TAG:NOM-REG-CABECERA-:TAG1:     PIC X(16).
  05 :TAG:INICIO-REGS-CABECERA-:TAG1: PIC 9(04).
  05 :TAG:FINAL-REGS-CABECERA-:TAG1:  PIC 9(04).
  05 :TAG:PARTE-TOTAL-:TAG1:          PIC 9(02).
  05 :TAG:PARTE-DECIMAL-:TAG1:        PIC 9(02).
  05 :TAG:NOMBRE-LOGICO-:TAG1:        PIC 9(04).
  05 :TAG:TIPO-NA-:TAG1:              PIC X(01).
  05 :TAG:TIPO-MU-:TAG1:              PIC X(02).
  05 :TAG:ENCONTRADO-:TAG1:           PIC X(01).
```

Gracias al uso de la propiedad TAG (etiquetas) de COBOL, podemos emplear un mismo fichero de definición o estructura de datos, sin tener "n" ficheros con nombres diferentes y la misma estructura interna de datos. Para ello solamente tendremos que sustituir el TAG por un prefijo o sufijo para que se adecue al tratamiento que más nos interese. El uso de este tipo de estructuras de COBOL nos ha facilitado en gran medida la programación del motor, ya que si requerimos hacer un cambio, lo hacemos una única vez y no tenemos que tocar en "n" ficheros que estuviesen implicados.

Tal y como se ha estructurado la información de los ficheros de especificación de los registros a leer y a escribir, que por una parte tenemos la definición de los tipos de datos y por otra parte existe un fichero que contiene la posición que ocupan cada variable dentro del fichero de datos que vamos a leer o escribir, esto nos permite que podamos definir estructuras de datos extensas tanto de entrada como de salida y luego solamente emplear aquellas variables que se consideren oportunas, lo que proporciona una gran flexibilidad, ya que si la

persona que nos pasa las estructuras de datos hace cambios con mucha frecuencia solamente nos tiene que indicar en que posición se encuentra la variable y no darnos toda la información del diccionario de datos, lo que puede incurrir en errores (tipo de datos numérico o alfanumérico, longitud de la parte entera o decimal). Y lo más importante, se separa la asociación de lo que llamaremos el nombre lógico del nombre físico. Este último punto es fundamental para el funcionamiento del motor de flujos, ya que el motor no trabaja con nombres físicos que se otorgan a las variables (ejemplo variable Estado Civil) sino que la herramienta trabaja internamente con unas codificaciones que hacen que la ligazón entre los ficheros de definición de cabeceras sea igual a la posición de las pilas o estructuras donde se almacenan los datos.

6.2 Estructura de datos

Al igual que tenemos dos estructuras de cabeceras entrada y salida, que sirven para definir cómo se tienen que leer los datos, tenemos una estructura con los datos de entrada (los cuales se leen en la versión de ficheros) y otro con los de salida (los que escribimos como resultado cuando trabajamos con ficheros).

La primera estructura, como su nombre indica contiene toda la información de los datos de entrada. Gracias a la estructura de cabecera de entrada, y después con la estructura de variables, podemos convertir los datos en variables de tipo numérico o alfanumérico detallando su longitud y precisión.

Una de las peculiaridades que posee esta estructura consiste, en que la interface puede contener información que llamaremos única y luego información de forma múltiple. Esto como ya hemos comentado con anterioridad, nos sirve por ejemplo para poder evaluar información de propuestas de créditos (Scorings) en las cuales pues una misma operación de préstamo puede tener "n" intervinientes.

Otra característica consiste en que dentro de la interface de datos existe una cabecera fija por registro que identifica cuál es la versión del motor con la que tiene que ejecutarse o evaluarse la información. A continuación detallamos esta cabecera que esta definida en la copy GDATIN:

5 caracteres que identifican el código de flujo con el cual queremos evaluar la información

5 caracteres que identifican la versión dentro del flujo que queremos usar para evaluar la información

2 caracteres reservados, actualmente no usados

1 carácter para indicar el nivel

3 caracteres que nos indican el número de bloques que vamos a leer.

2000 caracteres donde se codifican las diferentes variables.

A continuación se incluye la copy (GDATIN):

```
01 :TAG:REG-DATOS:TAG1:.
   03 :TAG:COD-FLUJO:TAG1:          PIC X(05).
   03 :TAG:COD-VERSION:TAG1:        PIC X(05).
   03 :TAG:RESERVADO:TAG1:          PIC 9(02).
   03 :TAG:NIVELES:TAG1:           PIC 9(01).
   03 :TAG:BLOQUESL1:TAG1:.
      05 :TAG:BLOQUEL1:TAG1:        PIC 9(03)
      OCCURS 1 TIMES.
   03 :TAG:REGS-DATOS:TAG1:.
      05 :TAG:REG-DAT:TAG1:         PIC X(01)
      OCCURS 2000 TIMES.
```

```
01 :TAG:REG-DATOS:TAG1:.
 03 :TAG:COD-FLUJO:TAG1:          PIC X(05).
 03 :TAG:COD-VERSION:TAG1:        PIC X(05).
 03 :TAG:RESERVADO:TAG1:          PIC 9(02).
 03 :TAG:NIVELES:TAG1:            PIC 9(01).
 03 :TAG:NUM-REGS-DATOS:TAG1:     PIC 9(09).
 03 :TAG:BLOQUESL1:TAG1:.
   05 :TAG:BLOQUEL1:TAG1:        PIC 9(03)
   OCCURS 1 TIMES.
 03 :TAG:REGS-DATOS:TAG1:.
   05 :TAG:REG-DAT:TAG1:         PIC X(01)
   OCCURS 2000 TIMES.
```

6.3 Fichero intérprete

Esta estructura es la que contiene toda la lógica o instrucciones que debe evaluar el Programa Intérprete.

Definición del contenido de la estructura:

-9 caracteres que nos indicaran el número de columnas o caracteres a leer.

-N caracteres que contendrán el flujo en función del número que contenga la variable anterior.

COBOL tiene una característica que ha sido fundamental a la hora de realizar este proyecto y consiste en que podemos definir la longitud de las estructuras de datos en tiempo de ejecución y en función de los datos que contiene la misma estructura, esto se consigue gracias a la instrucción OCCURS DEPENDING ON.

Para esta estructura ha sido fundamental contar con esta funcionalidad, a continuación se muestra el detalle:

```
01 :TAG:REG-TIRAPOLACA.
  03 :TAG:NUM-REGS-MODELO          PIC 9(09).
  03 :TAG:REGS-MODELOG.
    05 :TAG:REGS-MODELO           PIC X(01)
    OCCURS 1 TO 850000 DEPENDING ON
    :TAG:NUM-REGS-MODELO.
```

6.3.1 Tipología de funciones

A continuación hacemos mención a diferentes tipologías de funciones que hemos generado para que el motor sea lo más abierto posible y pueda servir para diferentes ámbitos (no solo para banca). Las funciones principalmente están divididas en 3 tipologías:

I. Numéricas
II. Lógicas
III. Texto

I. Funciones Numéricas:

- Suma -> Realiza la suma aritmética de 2 números
- Resta -> Realiza la resta aritmética de 2 números
- División entera - > Nos devuelve la expresión en forma entera del cociente entre dos elementos dividendo y divisor.
- Exponencial - > Nos devuelve el valor exponencial del número (n) que se le indique ($e^{n)}$
- Multiplicación: -> Nos devuelve el resultado de la multiplicación de 2 elementos o números.
- Entre -> Nos devuelve el cociente entre dos elementos dividendo y divisor.
- Residuo -> Nos devuelve el residuo o resto de la división entre dos elementos dividendo y divisor.
- Potencia -> Nos permite elevar un número a la potencia que se desee.
- Truncar -> Esta función nos devuelve la parte entera de un número real.
- Redondear -> Esta función nos permite pasar de una variable real a una entera redondeando el resultado de forma que el valor decimal que tuviese en origen modifique el último dígito de la parte entera del nuevo número.

Ejemplos:

Valor inicial = 15,65 Valor final = 16

Valor inicial = 15,23 Valor final = 15

- Raíz Cuadrada -> Esta función devuelve el resultado de realizar la raíz cuadrada de un número
- Logit -> Función estadística que se emplea en el ámbito de modelos financieros. Su expresión es:

(1 / (1 + exponencial $^{-n}$))

- Máximo -> Sobre 2 elementos de tipo numérico nos devuelve el mayor de los 2.

- Mínimo -> Sobre 2 elementos de tipo numérico nos devuelve el menor de los 2.
- Random -> Función que nos permite obtener un número aleatorio en función de una semilla dada.
- Log -> logaritmo en base e del argumento dado.

II. Funciones Lógicas

- Mayor -> Sobre 2 elementos de tipo numérico nos devolverá un valor booleano. En caso de que el primer elemento sea mayor que el segundo nos devolverá un 1, en caso contrario un 0.
- Menor -> Sobre 2 elementos de tipo numérico nos devolverá un valor booleano. En caso de que el primer elemento sea menor que el segundo nos devolverá un 1, en caso contrario un 0.
- MenorIgual -> Sobre 2 elementos de tipo numérico nos devolverá un valor booleano. En caso de que el primer elemento sea menor o igual que el segundo nos devolverá un 1, en caso contrario un 0.
- MayorIgual-> Sobre 2 elementos de tipo numérico nos devolverá un valor booleano. En caso de que el primer elemento sea mayor que el segundo nos devolverá un 1, en caso contrario un 0.
- Igual-> Sobre 2 elementos de tipo numérico nos devolverá un valor booleano. En caso de que el primer elemento sea igual que el segundo nos devolverá un 1, en caso contrario un 0.
- Diferente -> Sobre 2 elementos de tipo numérico nos devolverá un valor booleano. En caso de que el primer elemento sea igual que el segundo nos devolverá un 0, en caso contrario un 1.
- Negado -> Sobre una expresión que devuelva un valor booleano, nos devolverá el valor contrario al resultado de la expresión. Si la expresión devuelve un 0 la expresión negado la cambia a 1 y si la expresión devuelve un 1 la expresión negado la cambia a 0 (NOT).
- Y -> Esta expresión nos permite evaluar resultados booleanos, fundamentalmente la usaremos para las condiciones que se definan sobre reglas o bucles y nos permitirá concatenar expresiones de tipo booleano. La función Y nos devuelve el valor 1 si los datos lógicos de las expresiones que la componen se cumplen.
- O -> Esta expresión nos permite evaluar resultados booleanos, fundamentalmente la usaremos para las condiciones que se definan sobre reglas o bucles y nos permitirá concatenar expresiones de tipo booleano. La función O nos devuelve el valor 1 si alguno de los datos lógicos de las expresiones que la componen se cumplen.

III. Funciones de Texto.

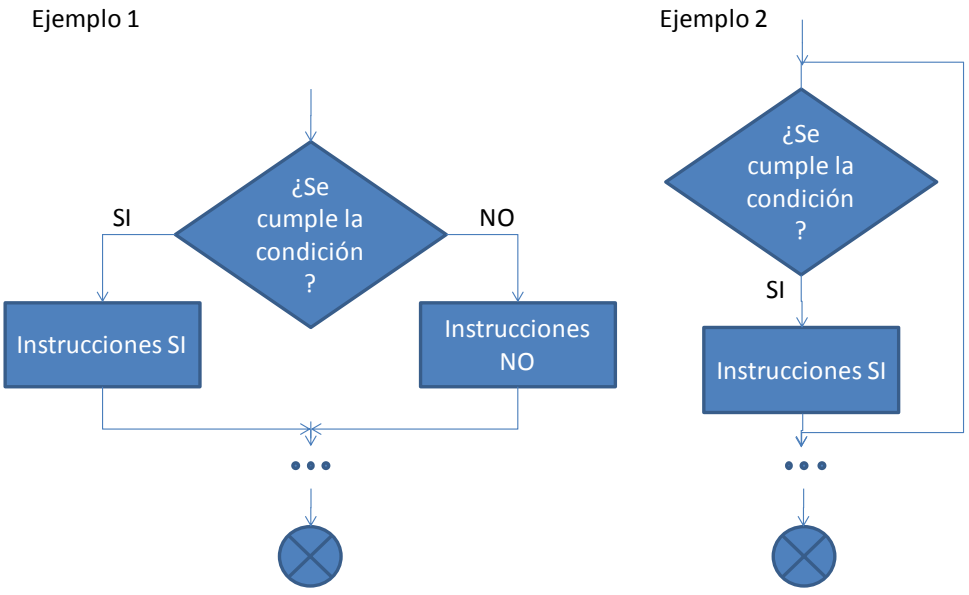
- Longitud - > Nos devuelve el número de caracteres que contiene una cadena de tipo string.
- Subcadena ->Nos permite extraer de una cadena de tipo string un trozo de la cadena, indicando la posición inicial y el número de caracteres que queremos extraer.
- Concatenar -> Nos permite anexar una cadena de tipo string a otra cadena de tipo string.

- No informado -> Esta funcionalidad se ha implementado adhoc para los modelos estadísticos financieros, ya que en muchas ocasiones se otorgan diferentes pesos o puntuaciones a variables que no han sido informadas y diferenciándolas de las que se han informado con valor, aun siendo cero. NOINF(variable) nos devuelve 1 o 0 en función de si esta informado o no.

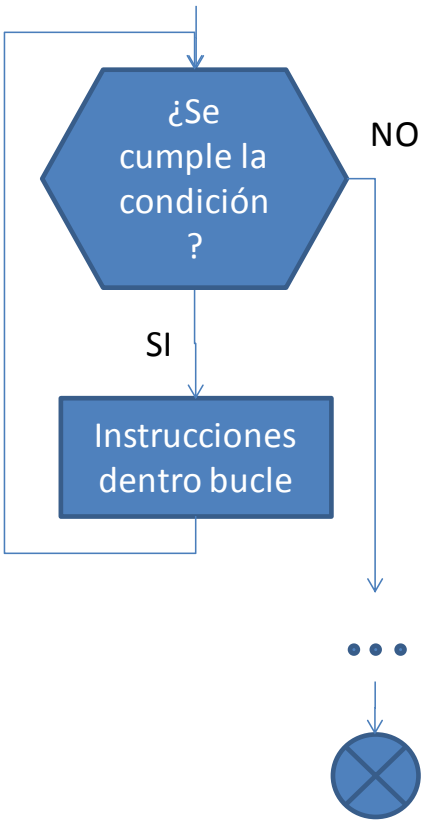
6.3.2 Tipología de funciones de flujos

Para poder llevar a cabo la construcción de flujos es necesario contemplar una serie de operaciones básicas para así poder implementar cualquier flujo que se defina. A continuación vamos a describir las funcionalidades diseñadas para poder llevar a cabo la implementación de todo tipo de flujos.

- REGLA: esta funcionalidad se trata de un condicional que en función de si se cumple o no la condición establecida (una o varias, ya sea usando las expresiones AND o OR) podemos decidir que instrucciones dentro de la misma se deban ejecutar. En el parámetro de condición podemos emplear cualquiera de los operandos o instrucciones que hemos visto en el punto anterior, permitiéndonos incluir unas dentro de otras. Se pueden dar reglas que contengan tanto instrucciones en el caso de que se cumpla la condición e instrucciones en el caso de que no se cumpla la condición (Ejemplo1) o solamente cuando se cumpla la condición (Ejemplo 2).



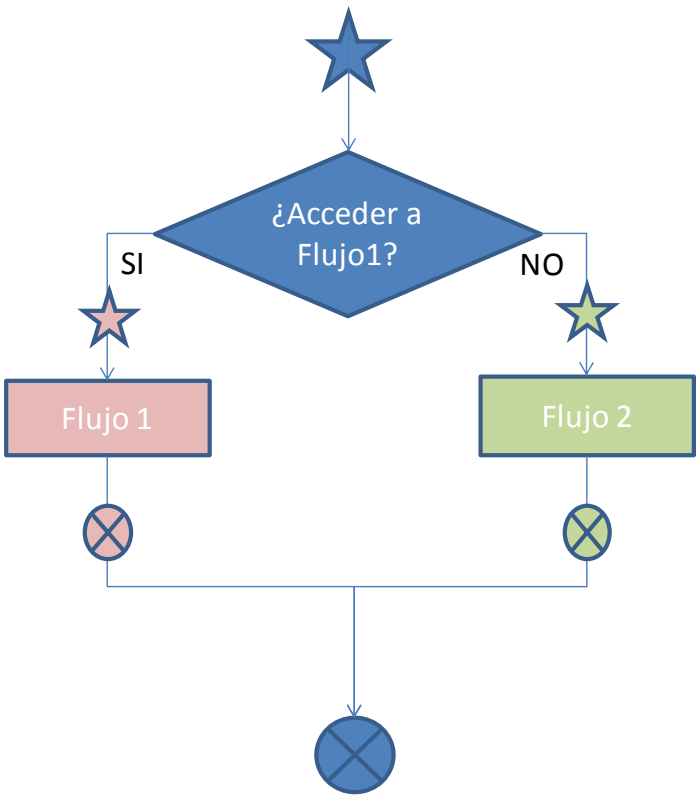
- BUCLE: Esta instrucción funciona de forma similar al Ejemplo 2 de la instrucción REGLA, la única diferencia radica en que se ejecutará "n" veces hasta que no se cumpla la condición. Esta funcionalidad nos permitirá evitar tener que repetir trozos de código en el cual el funcionamiento sea el mismo.



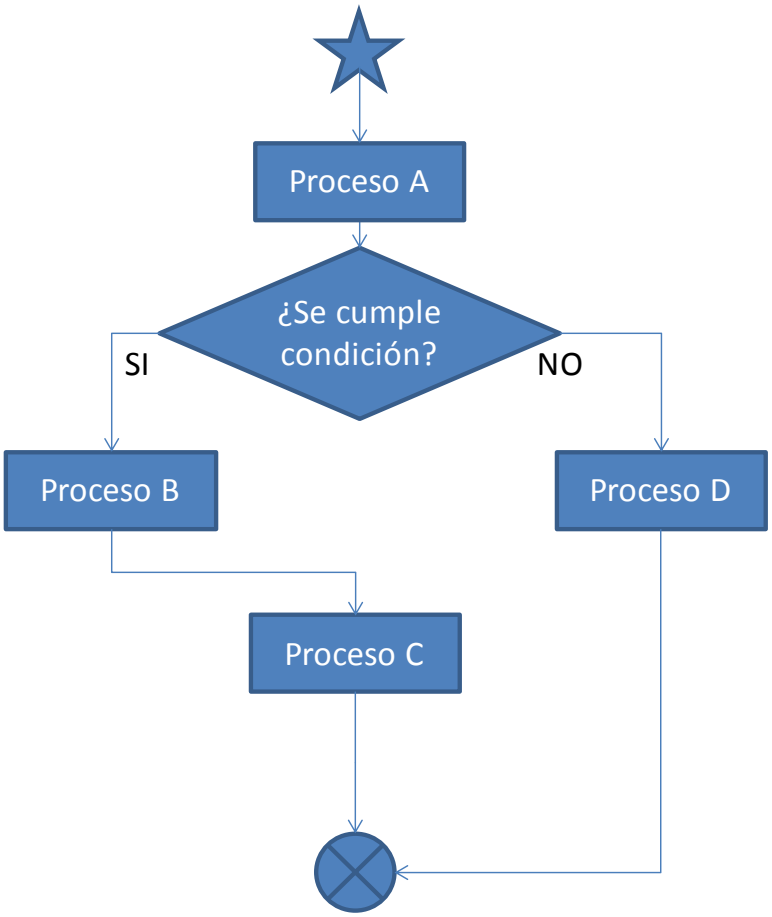
- ETIQUETA: Esta funcionalidad nos permite marcar dónde empiezan o terminan trozos de código que luego puedan ser invocados en cualquier momento de la ejecución del programa.

- SALTO: Esta funcionalidad nos permite movernos hacia las etiquetas que hayamos definido dentro del flujo. De esta forma podremos controlar por ejemplo los errores, o si tenemos subflujos dentro de un flujo principal, podremos movernos al punto donde este el subflujo al cual nos interese acceder.

Ejemplo 1:



Ejemplo 2:



6.3.3 Definición Tokens

En este apartado vamos a explicar cómo se han implementado cada uno de las funciones que se han explicado en los apartados anteriores. Esta explicación se va a realizar llegando a la mínima expresión qué es el token.

El token para este proyecto es la mínima expresión que vamos a evaluar, se compone siempre de la siguiente estructura:

V1,V2#

1)Donde V1 puede ser del tipo:

- Operador(SUMA, RESTA, DIVISION, etc.)

- Constante(entera, float, carácter, etc.)

- Variable (entera, float, carácter, etc.)

2)Para indicar que se ha terminado la definición del tipo de expresión se pone el símbolo coma ",".

3)A continuación viene V2 que podrá estar compuesto en el caso de que V1 sea:

- Operador –> nos indicará el tipo de operando a emplear, (suma, resta, etc.)
- Constante - > veremos directamente el valor que se ha indicado en el flujo. En caso de ser un número entero se representará con el signo seguido del valor numérico (no es necesario para los valores positivos indicar el signo). En caso de ser un float seguirá el mismo formato que los numéricos pero la parte decimal irá separada de la entera con el carácter punto ".". En el caso de que se trate de un string (cadena de caracteres) el valor irá incluido dentro de los marcadores &, por lo que si queremos poner el texto "Hola", la forma de representarlo en el código tiene que ser con &Hola&.
- Variable -> las variables se representarán directamente con el código de variable que se haya asignado previamente.

4)Por último el carácter almohadilla "#" representará el final de un token.

Ejemplos de tokens:

1) 7,=# Este token nos está indicando que es de tipo OPERADOR. Está indicando que el valor que contiene es el símbolo "=", que traducido al motor significa que se trata de una multiplicación.

2) 7,8# Este token también nos está indicando que es de tipo OPERADOR. Está indicando que el valor que contiene es el símbolo "8", que traducido al motor significa que se trata de una suma.

3) 5,1019# Este token nos está indicando que se trata de una variable de tipo carácter, por tanto el valor 1019 se corresponderá con el código asignado a una variable que contenga un valor de tipo alfanumérico (string).
4) *,&2&# Este token nos está indicando que se trata de una constante de tipo carácter, por tanto el valor &2& nos está indicando que el valor alfanumérico que contiene es un 2.

A continuación enumeramos los diferentes tipos de tokens que se pueden dar (parámetro V1) y todas sus posibles combinaciones (V2):

- 7,8# signo más (+)
- 7,9# signo menos (-)
- 7,:# división entera (/ nos devuelve la parte entera)
- 7,<# exponencial (e^x)
- 7,=# producto (*)
- 7,># entre (/ nos devuelve el resultado con decimales)
- 7,?# residuo (nos devuelve el residuo de una división)
- 7,@# potencia
- 7,A# mayor (>)
- 7,C# menor (<)
- 7,D# menor igual (<=)
- 7,X# mayor igual (>=)
- 7,E# igual (=)
- 7,F# diferente (<>)
- 7,G# negado (Not)
- 7,H# Y (AND)
- 7,I# O (OR)
- 7,J# truncado de un número
- 7,K# redondear
- 7,L# subcadena en strings
- 7,M# concatenar en strings
- 7,N# raíz cuadrada
- 7,O# longitud (nos indica cual es el largo de una cadena)
- 7,P# función LOGIT
- 7,Q# máximo
- 7,Y# mínimo
- 7,S# random (número aleatorio)
- 7,T# log (logaritmo neperiano)

Para poder llevar a cabo la implementación se tuvieron que sopesar diferentes tipos de datos que permitiese evaluar el motor o programa. Se vio necesario entre otras, diferenciar entre lo que llamaremos constantes y variables. Las constantes nos permiten la asignación de valores directamente sobre el código del flujo. En cambio las variables toman los datos de la información que se pasa por parámetro.

- (,VALOR# constante entera
-),VALOR# constante float (numérico con decimales)
- +,VALOR# constante logarítmica (con más precisión que el float)
- /,VALOR# constante dato
- *,VALOR# constante carácter

- 1,VARIABLE# variable entera
- 2,VARIABLE# variable float
- 3,VARIABLE# variable no informada
- 4,VARIABLE# variable otros
- 5,VARIABLE# variable carácter

A continuación vamos siguiendo el mismo orden que hemos visto en los apartados anteriores para mostrar cómo se construyen las funciones, ya que cada función requiere de diferentes número de parámetros (tokens). Para simplificar los ejemplos, vamos a emplear el valor 4 como primer parámetro, 2 como segundo parámetro , 3 como tercero y 5,55 como cuarto parámetro

- Suma (4+2) (2 parámetros)
  (,4#(,2#7,8#
- Resta (4-2) (2 parámetros)
  (,4#(,2#7,9#
- División entera (4/2) (2 parámetros)
  (,4#(,2#7,:#
- Exponencial e^4 (1 parámetro)
  (,4#7,<#
- Multiplicación (4+2) (2 parámetros)
  (,4#(,2#7,=#
- Entre (4/2) (2 parámetros)
  (,4#(,2#7,>#
- Residuo (4/2) (2 parámetros)
  (,4#(,2#7,?#
- Potencia (4^2) (2 parámetros)
  (,4#(,2#7,@#
- Mayor (4>2) (2 parámetros)
  (,4#(,2#7,A#

- Menor (4<2) (2 parámetros)
  (,4#(,2#7,C#
- Menor igual (4<=2) (2 parámetros)
  (,4#(,2#7,D#
- Mayor igual (4>=2) (2 parámetros)
  (,4#(,2#7,X#
- Igual (4=2) (2 parámetros)
  (,4#(,2#7,E#
- Diferente (4<>2) (2 parámetros)
  (,4#(,2#7,F#
- Negado (Not(4>2)) (2 parámetros)
  (,4#(,2#7,C#7,G#
- Y ((4>2) y (3<5,55)) (2 parámetros necesarios, ejemplo con 4 parámetros)
  (,4#(,2#7,A#(,3#(,5.55#7,C#7,H#
- O ((4>2) o (3<5,55)) (2 parámetros necesarios, ejemplo con 4 parámetros)
  (,4#(,2#7,A#(,3#(,5.55#7,C#7,I#
- Truncar (1 parámetro)
  (,5.55#7,J#
- Redondea (1 parámetro)
  (,5.55#7,K#
- Subcadena (3 parámetros)
  *,"HOLA MUNDO"#(,4#(,2#7,L#
- Concatenar (2 parámetros)
  *,"HOLA"# *,"MUNDO"#7,M#
- Raíz cuadrada (1 parámetro)
  (,4#7,N#
- Longitud
  *,"HOLA MUNDO"#7,O#
- Logit (1 parámetro)
  (,4#7,P#
- Máximo (2 parámetros)
  (,4#(,2#7,Q#
- Mínimo (2 parámetros)
  (,4#(,2#7,Y#
- Random (1 parámetro)
  (,4#7,S#
- Log (1 parámetro)
  (,4#7,T#

Cuando se finaliza una ristra de tokens (codificaciones que acaban en #), se tiene que marcar como que se ha finalizado la lectura de los tokens. La forma de realizarlo es a través del carácter especial "F".

Ejemplo:

*,"HOLA MUNDO"#(,4#(,2#7,L#F

Estas funciones que hemos visto anteriormente, pueden utilizarse en las siguientes tipologías de operaciones.

- Cálculos sobre Acciones
- Reglas
- Bucles

Cualquier tipo de operación que realicemos, tiene una instrucción de inicio y otra de fin. Esto es muy relevante para saber en todo momento que instrucción es la que esperamos.

A continuación vamos a explicar en detalle como se construyen cada una de las instrucciones que se puede realizar dentro del EWMR.

Instrucciones Variables y Acciones

El motor distingue de las instrucciones que están en el nivel principal, de las que están englobadas dentro de reglas o bucles. Las que no están englobadas, son las que llamamos instrucciones de variables, y las que están dentro las de "acción".

Las dos instrucciones "variables" o "acciones" nos sirven tanto para realizar asignaciones a variables , provenga el dato de una constante o directamente de otra variable, o para realizar cálculos sobre las mismas.

Todas las instrucciones de "variable" empiezan con una letra "V" y finalizan con una "v".

Ejemplos:

V1153),0.0#Fv Se está asignando a la variable 1153 una constante que toma por valor 0.0 (se está inicializando la variable). Su representación sería var1153 = 0.00

V11402,1016#Fv Se está asignando a la variable 1140 el contenido de otra variable (1016). Su representación sería var1140 = var1016

V10692,1093#2,1092#7,9#Fv Se está asignando a la variable 1069 el resultado del cálculo de la resta de 1093 y 1092. Su representación sería var1069 = var1093 – var1092

La única diferencia con entre la instrucción de variable y la instrucción de acción es que las instrucciones de "acción" tienen como cabecera una "A" y siempre tienen que terminar con una "a".

Ejemplo:

A11865,1152#(,0#(,1#7,L#Fa En esta instrucción se está realizando el siguiente cálculo con destino la variable 1186:

Var1186 = subcadena(var1152,0,1)

Instrucciones de Reglas

Las instrucciones de regla, como ya hemos comentado, nos permiten realizar acciones condicionales, y tener tratamientos diferenciados cuando se cumple la condición a evaluar o no. Las instrucciones de regla tienen como carácter inicial una "R" y como final una "r". En caso de que se contemple un tratamiento para cuando no se cumple la condición evaluada, el comienzo de la instrucción de esta instrucción se identificará con una "e".

- En primer lugar mostraremos una regla simple (sin tratamiento en caso de que no se cumpla la condición):

R(,300#2,1089#7,=#2,1092#7,A#FA10922,1065#(,300#2,1089#7,=#7,8#Far

El código anterior se representaría de la siguiente forma:

```
SI (300 * var1089) > var1092 ENTONCES
        var1092 = var1065 + 300 * var1089
FINSI
```

La descomposición por instrucciones sería

R(,300#2,1089#7,=#2,1092#7,A#F -> `SI (300 * var1089) > var1092 ENTONCES`
A10922,1065#(,300#2,1089#7,=#7,8#Fa -> `var1092 = var1065 + 300 * var1089`
r -> `FINSI`

- A continuación se muestra un ejemplo que contempla código cuando no se cumple la condición inicial:

R(,300#2,1089#7,=#2,1092#7,A#FA10922,1065#(,300#2,1089#7,=#7,8#Fae
A1091(,100#2,1082#7,=#Far

El código anterior se representaría de la siguiente forma:

```
SI (300 * var1089) > var1092 ENTONCES
        var1092 = var1065 + 300 * var1089
ELSE
        var1091 = 100 * var1082
FINSI
```

La descomposición por instrucciones sería

R(,300#2,1089#7,=#2,1092#7,A#F -> `SI (300 * var1089) > var1092 ENTONCES`
A10922,1065#(,300#2,1089#7,=#7,8#Fa -> `var1092 = var1065 + 300 * var1089`
E -> `ELSE`
A1091(,100#2,1082#7,=#Fa -> `var1091 = 100 * var1082`
r -> `FINSI`

Instrucciones de Bucles

Los bucles, son instrucciones similares a las reglas, pero no disponen de un bloque de instrucciones en el caso de que no se cumpla la condición principal. Si no se cumple la condición principal se terminan la iteración del mismo. El comienzo de esta instrucción se representa con una "B" y el final con una "b"

Para controlar los bucles y que no se realicen iteraciones de forma indiscriminada, se ha habilitado una variable que se le pasa por parámetro al bucle para evitar que estos hagan que la aplicación se quede colgada. Este será el primer parámetro de la función BUCLE y tiene 5 posiciones (con lo cual tendríamos un máximo de 99.999 iteraciones). Lo ideal es que la persona que programe el flujo piense un número razonable de iteraciones que deba realizar el bucle, pasado ese tope se considerará un error. La construcción del bucle, una vez pasados los 5 caracteres funciona de igual forma que las reglas.

Ejemplo:

B001002,1000#2,1092#7,C#A10002,1000#(,1#7,8#Fab

Su representación sería:

```
MIENTRAS (var1000 < var1092)
        var1000= var1000 + 1
FINMIENTRAS
```

Donde

B2,1000#2,1092#7,C# -> `MIENTRAS (var1000 < var1092)`
A10002,1000#(,1#7,8#Fa -> `var1000= var1000 + 1`
B -> `FINMIENTRAS`

Instrucciones de Etiquetas

La representación de esta instrucción es de las más triviales ya que solamente consiste en poner el carácter "Q" seguido de 3 dígitos para indicar el número de etiqueta.

Esta instrucción no se debe poner dentro de un bucle o regla, ya que no tendríamos cargada la pila de instrucciones y se podrían ocasionar comportamientos extraños si se accediese a la posición de dichas etiquetas.

Instrucciones de Saltos

Estas instrucciones nos permiten saltar o acceder a la posición donde se encuentre una etiqueta. La forma de codificar esta instrucción es también muy sencilla, la primera letra de la instrucción será una "S" e irá seguida de 3 dígitos que nos indicarán a que etiqueta queremos acceder o saltar.

Tanto las instrucciones de Etiquetas y Saltos al ser tan sencillas (son solamente 4 caracteres) no requieren de instrucción de finalización como sucede con las "variables", "acciones", "reglas" o "bucles".

Instrucción de Lectura

La instrucción de lectura se emplea para indicar (dentro del flujo) que se requiere realizar una lectura (sobre el fichero de datos del cual obtenemos la información o de la estructura que pasa la información por parámetro). De esta forma, en caso que tengamos parte del código que se tenga que ejecutar N veces en función de la información de entrada, nos permitirá refrescar el set de datos que estamos evaluando o ejecutando.

La forma de articular esta instrucción es igual que la de salto, pero en vez del carácter "S" para identificar la instrucción se hace con el carácter "U" seguido de 3 dígitos con los cuales identificaremos a la posición a la cual queremos saltar. La forma de controlar si se debe acceder o no al parámetro de entrada para leer la información, se hará en función de la evaluación de las variables IND-LECTURAS y MAX-LECTURAS.

Instrucción Otros

En los datos que se proporcionan a las herramientas en muchas ocasiones vienen variables que poseen catálogos muy extensos. Esto a la hora de programar un flujo o un modelo nos puede dificultar el proceso de escribir el código. Por ello se ha creado una funcionalidad que nos permite a través de un flag y desde el código, indicar si el valor que tiene actualmente la variable lo consideramos de la categoría otros, de forma que podremos darle un tratamiento adhoc, y no tendremos que picar en una regla todas sus posibles combinaciones.

A continuación se muestra un ejemplo de uso de la instrucción otros:

Ejemplo:
Imaginemos que tenemos un catálogo de profesiones y para nuestro flujo solo nos interese tener en cuenta a los profesores y los médicos, el resto de categorías les daremos un tratamiento diferente, pues para no picar todas las categorías diremos que si la variable esta informada y tiene valor diferente a maestros y profesores, la variable tiene valor OTROS, por tanto preguntando a la variable si el flag OTROS esta activado ya será suficiente para otorgarle el comportamiento que deseemos.

Esta instrucción tiene solo un parámetro que es el nombre de la variable, el primer carácter que identifica a la función otros es la letra "Z", seguida de 4 dígitos que identifican a la variable y un dígito adicional para indicar en caso de que sea 1 que se activa y 0 para desactivar el flag (Z10001)

Otra funcionalidad de esta instrucción consiste en que la podemos usar para controlar que todos los valores que se introducen en una variable sean válidos, es decir, que los tengamos contemplados.

Instrucción NoInformado

Las variables que se proporcionan a las herramientas en muchas ocasiones vienen sin datos (vacías). Es necesario por ello diferenciar esta casuística, sobre todo cuando hablamos de modelos, ya que normalmente no tienen que tener el mismo tratamiento una variable que no se haya informado, de otra que tenga un valor 0.

Ejemplo:
Imaginemos que tenemos que otorgar un préstamo a una persona. Quizás nos interese diferenciar si nos dice su edad como cliente de sí no nos la dice. Pero imaginemos que la edad como cliente es de 0 años, ¿no habría que puntuar mejor al que nos dice su edad, aunque sea 0 de la que no nos la dice?.

Esta función nos permite poder poner un flag a las variables para saber si vienen o no informadas de origen y poder discriminar este tipo de comportamientos.

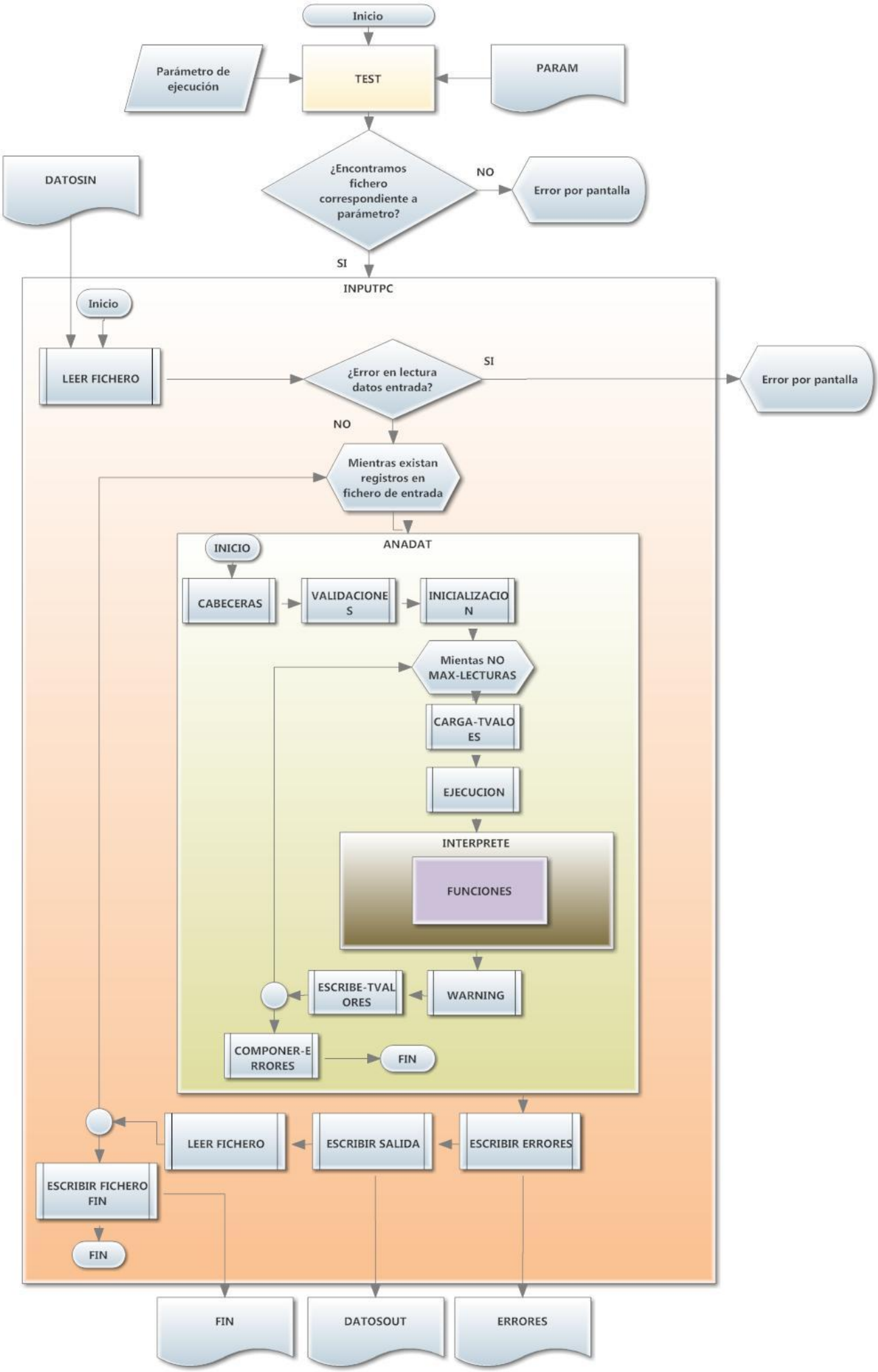
La forma de invocar a esta funcionalidad es como si se tratase de tipo de variable (entera, float, etc.), por lo que cuando queramos codificar esta instrucción sustituiremos el código del tipo de variable (1,2,5) por un 3. Quedando así la invocación 3,1000#.

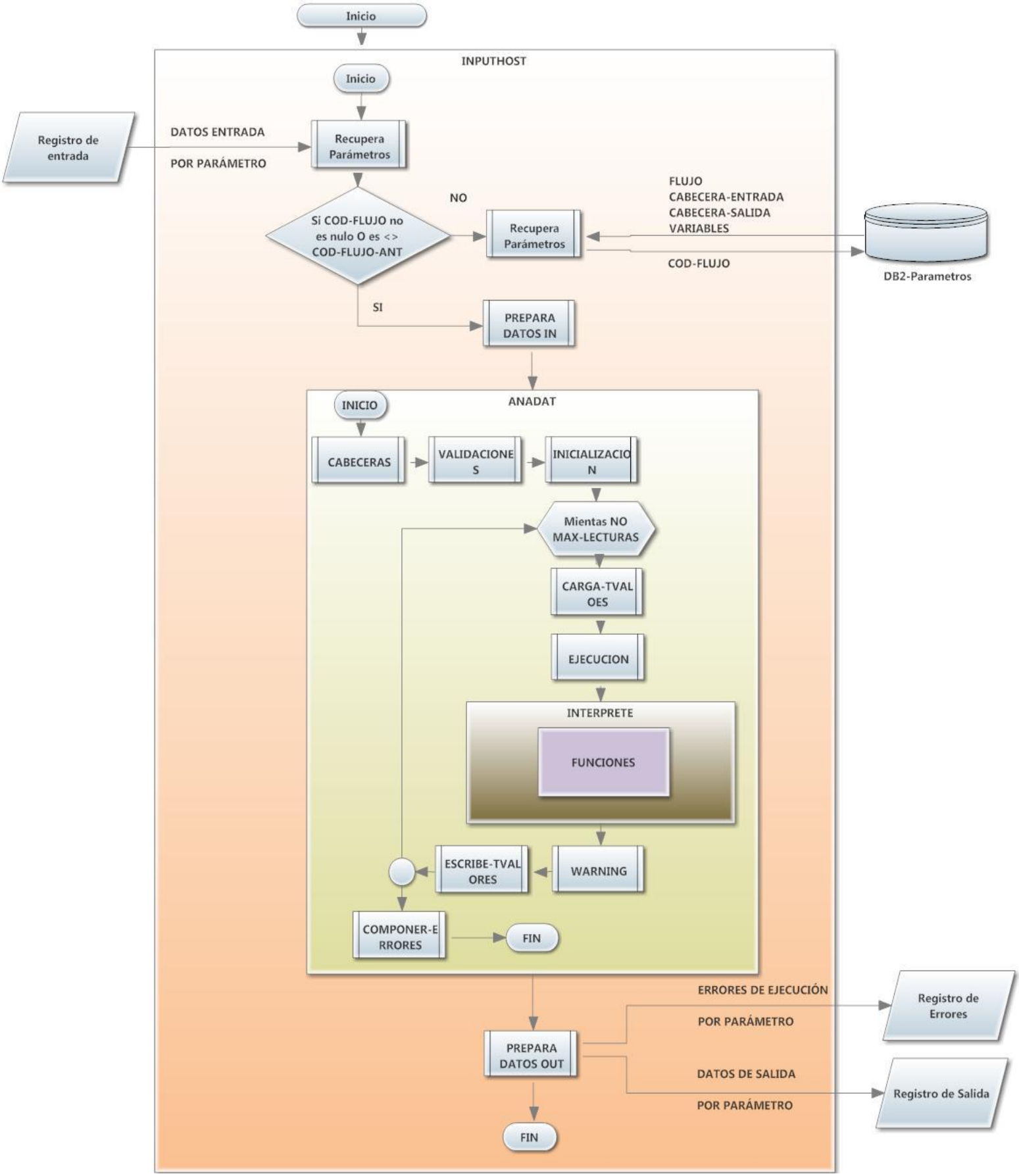
Instrucción de Fin de Flujo

Al permitirse los saltos dentro del flujo, se pueden dar el caso de que un flujo tenga que terminar a mitad de la ristra de instrucciones de la TiraPolaca. En consecuencia, es necesario crear un mecanismo para poder indicar si se ha producido un fin de flujo. La forma de representarlo es con el carácter "W".

6.4 *Diagrama de Clases u Objetos*

A continuación se muestran los diagramas de objetos que componen la herramienta, teniendo en cuenta en un primer lugar la versión PC y luego la versión HOST.


Inicio
Parámetro de ejecución
TEST
PARAM
DATOSIN
¿Encontramos fichero correspondiente a parámetro?
NO
Error por pantalla
SI
INPUTPC
Inicio
LEER FICHERO
¿Error en lectura datos entrada?
SI
Error por pantalla
NO
Mientras existan registros en fichero de entrada
ANADAT
INICIO
CABECERAS
VALIDACIONES
INICIALIZACION
Mientas NO MAX-LECTURAS
CARGA-TVALOES
EJECUCION
INTERPRETE
FUNCIONES
ESCRIBE-TVALORES
WARNING
COMPONER-ERRORES
FIN
LEER FICHERO
ESCRIBIR SALIDA
ESCRIBIR ERRORES
ESCRIBIR FICHERO FIN
FIN
FIN
DATOSOUT
ERRORES


Inicio
INPUTHOST
Inicio
Registro de entrada
DATOS ENTRADA
POR PARÁMETRO
Recupera Parámetros
Si COD-FLUJO no es nulo O es <> COD-FLUJO-ANT
NO
Recupera Parámetros
FLUJO
CABECERA-ENTRADA
CABECERA-SALIDA
VARIABLES
COD-FLUJO
DB2-Parametros
SI
PREPARA DATOS IN
ANADAT
INICIO
CABECERAS
VALIDACIONES
INICIALIZACION
Mientas NO MAX-LECTURAS
CARGA-TVALOES
EJECUCION
INTERPRETE
FUNCIONES
ESCRIBE-TVALORES
WARNING
COMPONER-ERRORES
FIN
ERRORES DE EJECUCIÓN
POR PARÁMETRO
Registro de Errores
PREPARA DATOS OUT
DATOS DE SALIDA
POR PARÁMETRO
Registro de Salida
FIN

6.5 *Modelo de BD*

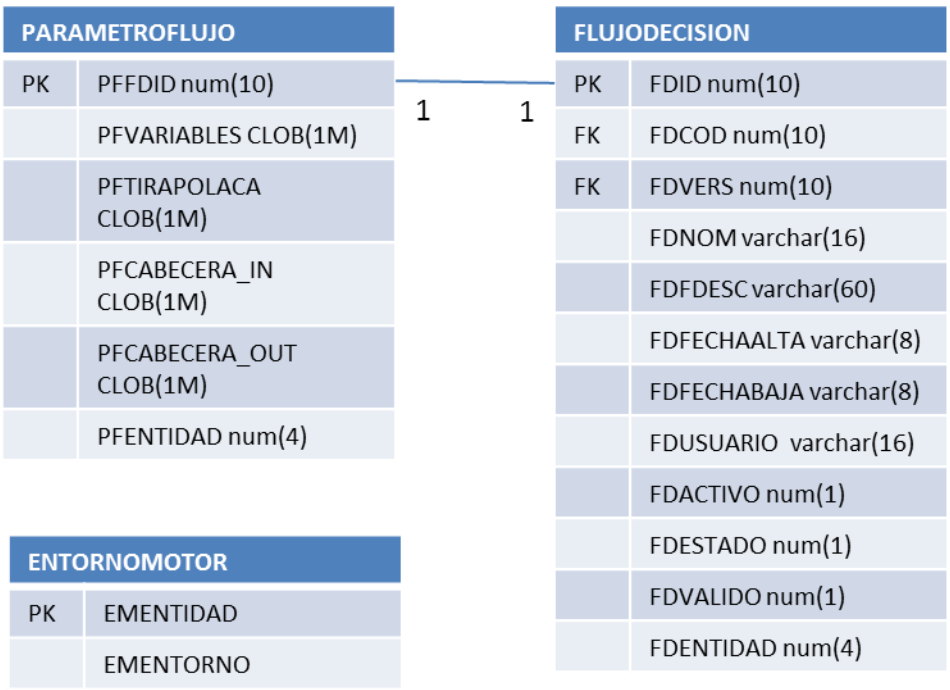
En este apartado se pretende explicar todas las funcionalidades que se han implementado para la versión Host estando conectado el motor a una BBDD.

A continuación se pueden ver los scripts de creación de tablas.

```
Scripts de creación de tablas DB2


DROP TABLE FLUJODECISION;
DROP TABLE PARAMETROFLUJO;

CREATE TABLE FLUJODECISION
(       FDID NUM(10) NOT NULL,
        FDCOD NUM(10) NOT NULL,
        FDVERS NUM(4) NOT NULL,
        FDNOM VARCHAR(16) NOT NULL,
        FDDESC VARCHAR(60),
        FDFECHAALTA VARCHAR(8),
        FDFECHABAJA VARCHAR(8),
        FDUSUARIO VARCHAR(16),
        FDACTIVO NUM(1) NOT NULL,
        FDESTADO NUM(1) NOT NULL,
        FDVALIDO NUM(1) NOT NULL,
        FDENTIDAD NUMERIC(4) NOT NULL,
        CONSTRAINT   PK_FLUJODECISION   PRIMARY   KEY
      (FDID),
        CONSTRAINT  UN_FLUJODECISION  UNIQUE  (FDCOD,
      FDVERS)
);

CREATE TABLE PARAMETROFLUJO
(
        PFFDID NUM(10) NOT NULL,
        PFVARIABLES CLOB(1M),
        PFTIRAPOLACA CLOB(1M),
        PFCABECERA_IN CLOB(1M),
        PFCABECERA_OUT CLOB(1M),
        PFENTIDAD NUMERIC(4) NOT NULL,
        CONSTRAINT   PK_PARAMETROFLUJO   PRIMARY   KEY
      (PFFDID),
        CONSTRAINT  FK_PFFDID  FOREIGN  KEY  (PFFDID)
      REFERENCES FLUJODECISION(FDID)
);
```

```
CREATE TABLE ENTORNOMOTOR
(
        EMENTIDAD NUMERIC(4) NOT NULL,
        EMENTORNO NUMERIC(1) NOT NULL,
        CONSTRAINT PK_EMOTOR PRIMARY KEY (EMENTIDAD)
);
```

Como se puede apreciar en el esquema ER (entidad relación) que se muestra a continuación, existen únicamente 3 tablas; PARAMENTROFLUJO, FLUJODECISION y ENTORNOMOTOR.



Inicialmente para el módulo HOST, se pensó en desarrollar unas tablas para almacenar la información de entrada y de salida de datos (DATOS_IN, DATOS_OUT y ERRORES), pero se vio que aunque su implementación era muy sencilla, carecía completamente de lógica ya que lo que no se podía hacer era construir unas tablas de datos de entrada y de salida, que tuviesen un formato preestablecido (copys GDATOSIN, GDATOSOU) y que el contenido fuese "una copia formateada" a las tablas que tuviese el cliente, las cuales contuviesen la información con los formatos reales.

Con esto, lo que pretendo explicar, es que normalmente el cliente va a tener unos sets de datos en unos formatos ya definidos (tablas, estructuras, etc.), ya

puedan ser en otros entornos diferentes al nuestro u otros sistemas de BBDD, los cuales nos tienen que facilitar para que funcione el programa. No tiene sentido, construir una estructura o programa que vaya a buscar la información a las fuentes origen, ya que pueden ser muy diversas y variadas, y que posteriormente se almacene dicha información en unas tablas del sistema EWMR, una vez transformados los datos.

Ejemplo:

Imaginemos que vamos a instalar el producto en la empresa EXPLOTACIÓN DE FLUJOS. Esta empresa tiene una BD en entorno ORACLE, y en cambio nuestras especificaciones iniciales son que la BD sea DB2. El programa habría que adaptarlo levemente (cambiando los tipos de datos) para que funcionase en el entorno ORACLE. Pero imaginemos que lo que "vendemos al cliente" es que se trata de un producto cerrado y que tiene que ser DB2 la que almacene la información de los flujos. Tendríamos que construir un módulo que fuese a buscar la información al entorno ORACLE y lo transformase al entorno DB2, pero además se tendría que transformar la información al formato que requiere el aplicativo.

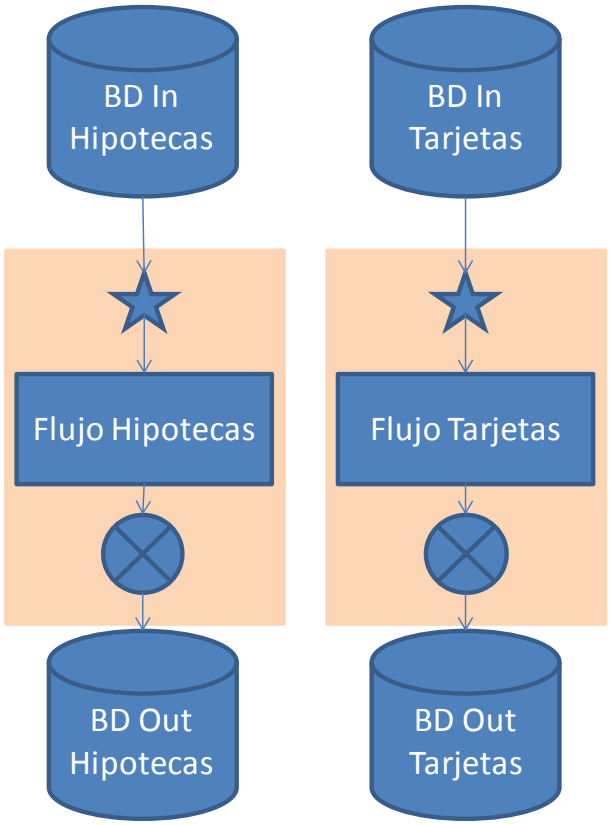
Otra situación, sería que el cliente tuviese el mismo sistema de BBDD (DB2). Igualmente tendríamos que realizar un módulo que fuese a buscar la información a la tabla en la cual se disponga de la información, por lo tanto ya estaríamos modificando los programas, ya que cambiaríamos los parámetros de conexión, etc., y además se tendría que realizar un módulo para transformar sus tipos de datos en los que requiere la aplicación.

Por dichos motivos, se descartó realizar este tipo de procesos y lo que se pensó fue en una estructura que permaneciese invariable en función del sistema en el cual se instale. Por lo cual, el resultado fue que la información de datos de entrada, salida y errores se manipulase a través de parámetros, con lo cual nos desligamos completamente de la infraestructura o tecnología con la que cuente el cliente, ya que será este mismo el que tenga que desarrollar las interfaces necesarias para poder transmitir la información al programa.

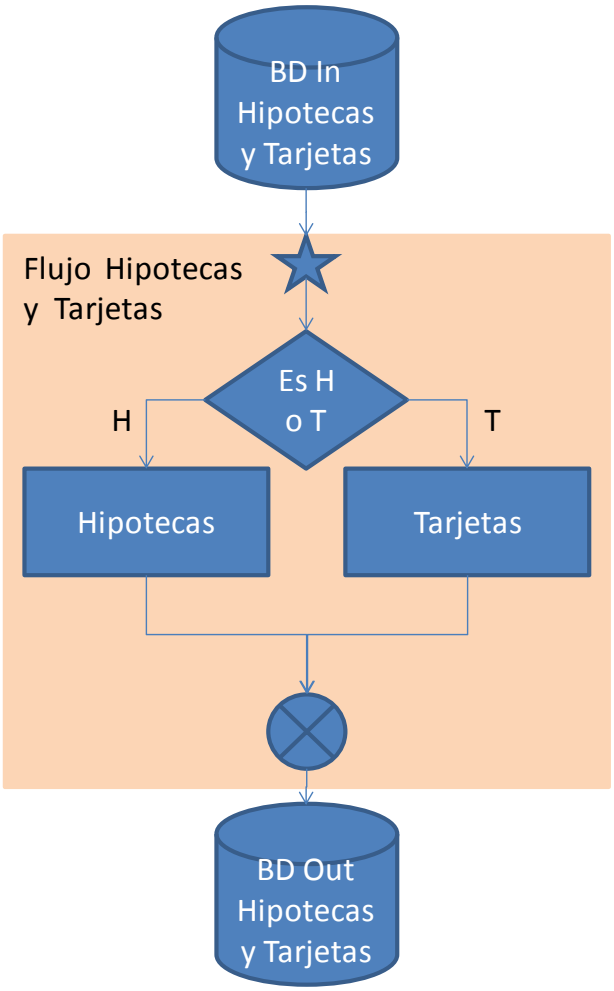
Para dotar a la herramienta de más funcionalidad, ésta posee la particularidad de que los datos a evaluar o procesar, tienen que proporcionarnos el código y versión del flujo con el cual se quiere evaluar cada uno de los registros. Esto se realiza de esta forma para así poder contemplar que tengamos una BBDD con toda la información, pero clasificada en función de posibles carteras diferentes, y así poder ejecutar los flujos en función de la tipología de la información.

Ejemplo:

Imaginemos que tenemos en una tabla los registros de clientes de productos bancarios de hipotecas y en otra tabla los de tarjetas. Estos registros serían evaluados cada uno con el flujo correspondiente en función del código y versión indicado.



No obstante podríamos tener un única BD que contuviese la información de Hipotecas y Tarjetas, y en función de la información, y tras pasar por el flujo, se tomase la decisión de con qué subflujo se quiera evaluar. La única pega es que el flujo sería mucho más largo y por tanto también más complicado.



En conclusión los Flujos Independientes son más complejos en la realización de las llamadas, pero más sencillos en el funcionamiento y adicionalmente las estructuras y ficheros son más pequeños (ficheros de variables, cabeceras, flujo, etc..) . En cambio, en los flujos únicos es más sencillo el proceso de invocación a la herramienta, pero más complejos los ficheros que necesita (ya que el código de flujo será más largo y requerirá seguramente de más variables definidas en las cabeceras y fichero de variables).

Módulo Entrada Host

El Módulo INPUTHOST es un módulo que viene a ser muy parecido al que se ha generado para la lectura de los ficheros en la versión PC, pero que en vez de obtener parte de la información de los ficheros lo hace accediendo a la BBDD Host, y extrayendo la información en función de los parámetros que se indican en cada uno de los registros que contienen los datos de entrada y que son pasados por parámetro. La información que extraemos de la BD es la que se ha comentado en el punto anterior y es la referente a las CABECERAS(entrada y salida), VARIABLES y FLUJO.

Este módulo en un primer lugar establecerá la conexión con la BBDD, para posteriormente extraer la información de las estructuras, cerrando posteriormente la conexión con la BBDD. Una vea que hemos obtenido la información , se encargará de cargar las interfaces en el formato que necesita el módulo ANADAT, y una vez cargadas, invocará al módulo ANADAT.

El módulo INPUTHOST será invocado tantas veces como registros contenga el fichero de datos de entrada, para posteriormente ser evaluados. Mientras el código y la versión de flujo sean los mismos que los de la evaluación anterior, no se ejecutarán los pasos correspondientes a la BBDD (conexión, consulta de parámetros y desconexión), ya que estos datos ya los tendrá en memoria el módulo INPUTHOST, con lo cual el rendimiento será más óptimo, ya que nos evitaremos realizar consultas y las conexiones hacia la BBDD.

Módulo Actualizador de Parámetros.

Para llevar a cabo la administración de los flujos que se almacenan en la BD-HOST se creó el módulo ACTPAR (actualización de parámetros) que nos permite dar de alta nuevos flujos, actualizarlos, darlos de baja, o ponerlos como inactivos.

La información de entrada que evalúa el motor Host (inputhost) tiene que indicar el código y versión de flujo con el cual queremos que sea evaluada. Por ello, el flujo a extraer tiene que tener el estado = 2 (en caso de estar en entorno de producción) y la variable activo = 0. Cuando se va a producir una evaluación de un flujo, se obtiene a través del código y la versión la información de la BBDD. Si posteriormente existen más registros a evaluar y tienen el mismo código y versión de flujo, no se realizarán lecturas sobre la BD para obtener la información del flujo, ya que se dispone de la misma en memoria. En caso contrario, se realizará una nueva consulta a la BD para obtener la información correspondiente.

Tenemos dos posibles estados: 1 y 2 (desarrollo y producción). Los posibles valores de la variable activo son: 0 y 1 (activo e inactivo) . Adicionalmente existe una variable contenida en la tabla ENTORNOMOTOR que nos indica si el entorno en el que estamos trabajando es producción o desarrollo, ya que el comportamiento de actualización de las otras tablas es diferente. Tomará valor 0 para indicar que es desarrollo y 1 para indicar que es producción.

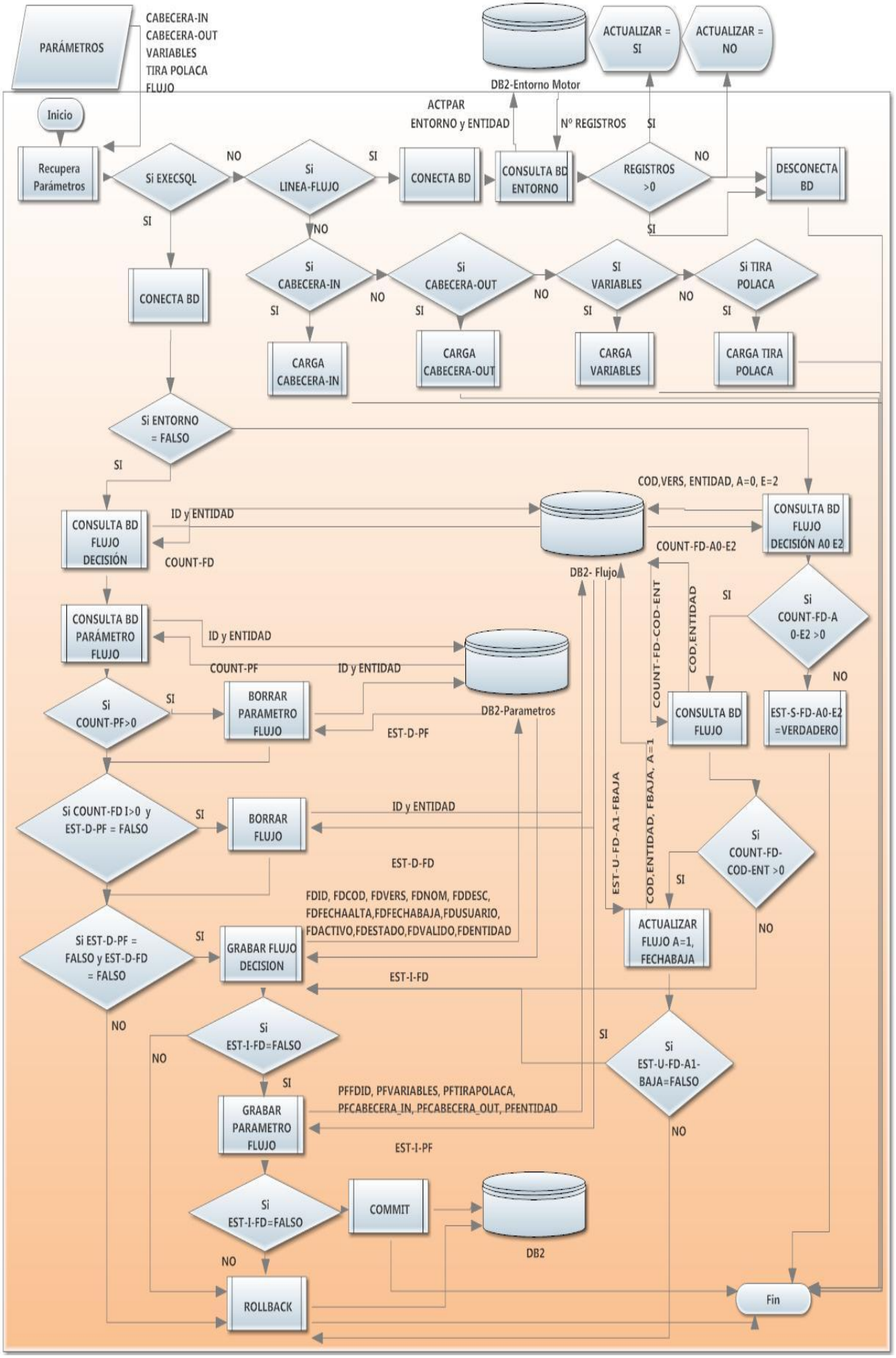
En caso de que queramos dar de alta un nuevo flujo y exista para la entidad e identificador, se dan dos posibles opciones de trabajo:

-Desarrollo. Se comprobará si existe. En caso de que exista se borrará y posteriormente se dará de alta.

-Producción. Se comprobará si existe. En caso de que exista el flujo, se pondrá con estado activo = 1 (inactivo) y se indicará la fecha de baja. A continuación se grabará el flujo de decisión y se incluirá un nuevo registro en la tabla parámetro flujo.

Esta diferenciación del tratamiento a aplicar se realiza de esta forma ya que se entiende que en el entorno de pruebas o desarrollo, estamos realizando cambios de forma continua. En cambio en el entorno de producción, no se pueden eliminar flujos con los cuales se hayan evaluado registros y por ello no se elimina, simplemente se cambia su estado, ya que podríamos llegar a usar un flujo dado de baja, pero realizando los cambios vía mantenimiento.

En el esquema que se muestra a continuación se puede ver el esquema o flujo que tiene dicho programa.


PARÁMETROS
CABECERA-IN
CABECERA-OUT
VARIABLES
TIRA POLACA
FLUJO
DB2-Entorno Motor
ACTUALIZAR = SI
ACTUALIZAR = NO
Inicio
ACTPAR
ENTORNO y ENTIDAD
Nº REGISTROS
SI
Recupera Parámetros
Si EXECSQL
NO
Si LINEA-FLUJO
SI
CONECTA BD
CONSULTA BD ENTORNO
REGISTROS >0
NO
DESCONECTA BD
SI
NO
CONECTA BD
Si CABECERA-IN
Si CABECERA-OUT
SI VARIABLES
Si TIRA POLACA
CARGA CABECERA-IN
CARGA CABECERA-OUT
CARGA VARIABLES
CARGA TIRA POLACA
Si ENTORNO = FALSO
COD,VERS, ENTIDAD, A=0, E=2
CONSULTA BD FLUJO DECISIÓN
ID y ENTIDAD
COUNT-FD
DB2- Flujo
CONSULTA BD FLUJO DECISIÓN A0 E2
COUNT-FD-A0-E2
CONSULTA BD PARÁMETRO FLUJO
ID y ENTIDAD
COUNT-PF
DB2-Parametros
Si COUNT-FD-A 0-E2 >0
COUNT-FD-COD-ENT
COD,ENTIDAD
Si COUNT-PF>0
BORRAR PARAMETRO FLUJO
EST-D-PF
CONSULTA BD FLUJO
EST-S-FD-A0-E2 =VERDADERO
Si COUNT-FD I>0 y EST-D-PF = FALSO
BORRAR FLUJO
ID y ENTIDAD
EST-D-FD
EST-U-FD-A1-FBAJA
COD,ENTIDAD, FBAJA, A=1
Si COUNT-FD-COD-ENT >0
FDID, FDCOD, FDVERS, FDNOM, FDDESC, FDFECHAALTA,FDFECHABAJA,FDUSUARIO, FDACTIVO,FDESTADO,FDVALIDO,FDENTIDAD
Si EST-D-PF = FALSO y EST-D-FD = FALSO
GRABAR FLUJO DECISION
ACTUALIZAR FLUJO A=1, FECHABAJA
EST-I-FD
Si EST-I-FD=FALSO
Si EST-U-FD-A1-BAJA=FALSO
PFFDID, PFVARIABLES, PFTIRAPOLACA, PFCABECERA_IN, PFCABECERA_OUT, PFENTIDAD
GRABAR PARAMETRO FLUJO
EST-I-PF
Si EST-I-FD=FALSO
COMMIT
DB2
ROLLBACK
Fin

7. Implementación

7.1 TEST

Este es un módulo que se implementa exclusivamente para la realización de las pruebas y así poder probar la ejecución de forma más sencilla desde el entorno PC. Nos permite entre otras cosas saber el tiempo de proceso que ha tardado en ejecutar las instrucciones o flujos que se han pasado por parámetro.

Para la realización de la demo se tendrá que ejecutar el fichero EWMR.EXE y seguido la cabecera del módulo PARAM que contendrá entre otras:

- La definición del flujo (información de identificación y descripción del flujo)
- La tira polaca (instrucciones a ejecutar)
- El módulo de definición de variables
- Cabecera de entrada
- Cabecera de salida

Ejemplo de ejecución: EWMR.exe 20120102PRUEBA

Ver código módulo TEST en el Anexo I y II, apartado 2.1.

Las funciones que dispone este módulo de programación son:

- SIMULACION, en un primer lugar invoca a la función TIEMPO-I, posteriormente a través del parámetro que se usa en la invocación del programa, se encarga de abrir el fichero de parámetros y realizar las lecturas de la información que contiene (cabeceras, tira instrucciones, variables). Posteriormente llama al módulo INPUTPC y TIEMPO-F.
- TIEMPO-I, devuelve la hora actual
- INPUTPC, se encarga de invocar al módulo COBOL INPUTPC
- TIEMPO-F, devuelve la hora actual y la diferencia con la hora inicial, así podremos saber el tiempo que ha tardado la ejecución.

7.2 Módulo de ejecución desde PC

Este módulo es el encargado de obtener del fichero que contiene los datos de entrada la información para que los evalué o procese el módulo ANADAT que es llamado desde este módulo. A su vez, una vez ejecutado el motor, será este módulo también el encargado de escribir en el fichero de salida el resultado de la ejecución, así como escribir en el fichero de errores (en caso de que se diesen, contendrá los códigos de error correspondientes) y un fichero que nos indicará que el programa ha concluido de forma correcta. Este fichero de status de finalización es necesario que se genere ya que como el motor permite evaluar más de un flujo y n registros de entrada, se podría dar el caso de que ejecutase

correctamente una parte y otra no.

Las funciones que contiene este módulo son:
- FICHEROS. Este módulo se encarga de abrir el resto de ficheros que se van a emplear (datos de entrada, salida, errores y fichero de ejecución correcta)
- DATOSC. Esta función transforma los datos obtenidos por parámetros de las cabeceras y variables a la estructura de datos que requiere el módulo ANADAT. Se realizará tanto para las cabeceras de entrada como de salida.
- DATOSOU. Se encarga de volcar ciertas variables relevantes de la ejecución como son el código de flujo versión, niveles, número de bloques y cantidad de registros de salda, a una estructura que no se borrará al volver a invocar al programa INPUTHOST, de forma que podremos saber cuáles son los valores que existían en una posterior ejecución.
- FLUJO. Esta función es la que invoca al módulo ANADAT, pasándole toda la información que requiere por parámetro.

Ver código módulo INPUTPC en el Anexo I y II, apartado 2.3.

7.3 Módulo de Análisis de datos

Este es uno de los módulos principales y más importantes del programa. Este módulo se encarga de transformar la información obtenida por parámetro (cabecera-in, cabecera-out y variables) en las estructuras de datos que necesitará el módulo INTERPRETE. Con la información de las 3 estructuras antes mencionadas, y obteniendo la información de la estructura de DATOS_IN se transforman la ristra de caracteres de entra en variables, en función de las longitudes y tipos de datos que se han definido en cabeceras y variables. Otra función importante que realiza este módulo consiste en la realización de una lectura rápida sobre la estructura que contiene la tira de instrucciones (tira polaca), para así poder almacenar la posición de cada una de las etiquetas que se hayan declarado. Una vez ejecutado el módulo INTÉRPRETE, este módulo realiza el paso inverso, convierte la información contenida en las estructuras internas para generar así la información de la estructura DATOS_OUT y ERRORES.

Se compone de las funciones que se enumeran a continuación:

- ANADAT. Esta es la función principal del módulo y es la que se encarga de llamar al resto de funciones que se muestran a continuación.
- CABECERAS. Esta función se encarga de llamar a las funciones CABECERAS-IN y CABECERAS-OUT
- CABECERA-IN. Esta función se encarga de revisar toda la estructura de datos de cabecera de entrada, revisando que la información que contiene sea la correcta. Por ejemplo, que las variables numéricas lo sean. Se encarga también de calcular parte del tratamiento de los registros múltiples o simples.

- CABECERA-OUT. Esta función en parte se encarga de realizar validaciones sobre la estructura de cabeceras de salida y calcular el tratamiento de registros múltiples o simples.
- ESCRIBE-TVALORES. Esta función se encarga de extraer de las estructuras que maneja el aplicativo los valores que contiene, ya sea en formato numérico, float o carácter y transformarlos a la interface de comunicación que invoca al módulo ANADAT, es decir, en formato texto. Para ello empleará los registros de CABECERA-OUT y variables una vez validados.
- CARGA-TVALORES. Esta función realiza justamente las instrucciones contrarias que la función ESCRIBE-TVALORES. Obtiene la información que viene por parámetro y gracias a la información que se ha validado en el módulo CABECERA-IN y VARIABLES es capaz de transformar la interface de entrada en las estructuras que luego manejaremos para poder realizar los cálculos.
- INICIALIZACION. En un primer lugar, esta función se encarga de calcular para las variables de cálculo artificiales o de salida la longitud máxima de las variables, para así cuando el módulo INTÉRPRETE realice los cálculos se pueda ver si la operación de destino va a provocar desbordamiento. También se usa para inicializar FLAGS y los NO INFORMADOS
- EJECUCION. Este módulo es el encargado de invocar al módulo INTÉRPRETE que es el que contiene las reglas, bucles, cálculos, etc..
- VALIDACIONES. Este módulo se encarga de comprobar sobre el fichero de variables que éste cumpla todas las condiciones, que la variable tipo de dato sólo contenga el valor numérico o alfanumérico, que las variables alfanuméricas no tengan informada la parte decimal, que la parte decimal no sea mayor que la parte total, etc…
- CARGA-SALTOS. Esta función se encarga de buscar en la tira polaca los posibles puntos de salto que indican dónde están ubicados los saltos no lógicos de los flujos.
- WARNING. Esta función es la encargada de almacenar los mensajes de advertencia que se hayan producido en la ejecución del intérprete pero que no hayan ocasionado errores graves sobre las variables empleadas. Por ejemplo, que existan variables que contengan otros valores a los contemplados en la ejecución, también se valida información sobre los no-informados, etc..
- COMPONER-ERRORES. Esta función traduce la estructura que almacena los errores hacia la estructura de salida que se emplea como interface. Además de los errores que hayan provocado el paro de la ejecución, también incluirá un apartado en el que figuren los WARNING.

Ver código módulo ANADAT en el Anexo I y II, apartado 2.3.

7.4 Módulo de Ejecución de Flujos y Reglas, intérprete de condiciones

Este es el módulo principal que se encarga de procesar las instrucciones de flujos y cálculos. Es la que interpreta carácter a carácter la tirapolaca y, a través de sus múltiples funciones la va descomponiendo, validando y ejecutando al mismo tiempo, token a token y función a función. Este programa es totalmente recursivo, una función puede invocar a otra y luego otra puede invocar a la primera y así. Justamente la inteligencia y a su vez complejidad de este programa radica en ello, ya que en determinadas situaciones es complicado saber en que llamada estamos, si en la primera , una intermedia o la final. Se ha evitado en la medida de lo posible tener código repetido, por ello a veces nos podemos encontrar funciones muy simples, pero que son invocadas con mucha frecuencia. En este módulo ha primado la sencillez y se ha intentado realizar un código que sea muy comprensible.

A continuación enumeramos cada una de las funciones internas de las que dispone:

- INTERPRETE-MAIN. Esta función principalmente se encarga de inicializar las estructuras que emplea la herramienta por cada ejecución. Se encarga también de invocar a la función EJECUTA.
- EJECUTA. Se podría decir que esta función es la encargada de llamar al resto de funciones que veremos a continuación. Se encarga del análisis a más alto nivel de la instrucción que se está evaluando, ya sea una REGLA, VARIABLE, BUCLE, SALTO, ETIQUETA, directiva de lectura, OTROS, directiva de error, u otros tipologías de datos no esperados (considerados por tanto errores). Esta función se tiene que ejecutar mientras no sea un findecadena, no exista directiva de fin, no se haya forzado la finalización o existan registros a leer de instrucciones.
- LEER-POLACA. Esta función se encarga de la descomposición de las instrucciones de REGLAS, BUCLES o VARIABLES. Una vez identificado el tipo de instrucción a procesar, se encarga de obtener todos los tokens y cargarlos en las diferentes estructuras de información hasta encontrar una directiva de fin de token "#".
- CALC-VARIABLE. Esta función se encarga de EVALUAR (calcular ristra de tokens) haciendo uso de las funciones I-EJECUTA, INDICE, PONER-DATO, D-EJECUTA e IDENT.
- CALC-BUCLE. Se encarga de marcar dónde empieza el bucle y dónde finaliza. Esto es importante, ya que podemos tener bucles dentro de bucles y hay que saber dónde empieza y finaliza cada uno de ellos, el principal y los que contiene. Otro parámetro que también se encarga de almacenar es el número máximo de vueltas permitido a un bucle, muy importante, ya que si la persona que diseña el bucle no tiene suficientes conocimientos de programación podría provocar bucles infinitos. Se encarga también de invocar a las funciones EVALUAR, I-EJECUTA y CALC.

- CALC-REGLA. El funcionamiento es el mismo que para los bucles, pero no contaremos con el parámetro de MAXVUELTAS. Invocará por tanto a EVALUAR, I-EJECUTA y CALC.
- CALC. Este es el módulo o función que se encarga de poder hacer realidad la anidación de reglas dentro de reglas, bucles con reglas o bucles con reglas y bucles (cualquier combinación que se nos ocurra) y que estas a su vez contengan ACCIONES, SALTOS, OTROS. Esta función hace uso de LEER-POLACA, EVALUAR, I-EJECUTA, IDINDEX, INDICE, PONER-DATO, IDSALTO, IDMAXVUELTAS, D-EJECUTA, IDENT.
- EVALUAR. Este módulo en función de las estructuras que contiene los tokens, se encarga de volcarlos a unas variables auxiliares que serán las empleadas para realizar los cálculos. Invoca a los métodos PUSH-PILA, SACAR-DATO, NOINF, OTROS y OPERAR.
- NO-INF. Esta función se encarga de devolver si una variable esta o no informada en función del flag informado asociada a la misma.
- OTROS. Esta función se encarga de devolver si el valor de la variable pertenece al grupo de otros, según el flag asociado a la misma.
- PUSH-PILA. Esta función se encarga en caso de que la variable a evaluar sea una constante o variable de cargar la información en una pila que usaremos para operar.
- GET-TOKEN. Esta función se encarga de evaluar cada uno de los tokens, principalmente su función es la de validarlos, que no nos encontremos con caracteres o instrucciones no esperados dentro de un token. Invoca al método TIPTOKEN.
- OPERAR. Esta es la función que se encarga de la realización de los cálculos matemáticos, booleanos o de texto de la herramienta, en la que están implementadas todas las instrucciones. Invoca a los métodos POP2, POPP y PUSHP. También se encarga de invocar al módulo COBOL FUNCIONES.
- POPP. Se encarga de obtener un dato, desapilando la información de la pila que almacena los cálculos que contiene un token.
- PUSHP. Se encarga de almacenar o apilar el resultado de una ejecución.
- SACAR-DATO. Esta función se encarga de volcar en variables auxiliares el contenido de los vectores que almacenan las variables numéricas para luego poder invocar al método PUSH-PILA.
- PONER-DATO. Esta función se encarga de volcar el resultado que contiene la pila de ejecución en las variables que deban contener el dato resultante del cálculo. Esta función valida el valor obtenido a la hora de volcarlo a la variable y verifica que no se produzca overflow por la parte entera.
- IDENT, TIPTOKEN, IDINDEX, IDMAXVUELTAS, IDSALTO son una serie de funciones que se encargan de leer un número de caracteres determinados e incrementar su índice correspondiente.
- INDICE. Esta función se encarga de devolver el nombre lógico de una variable existente en las estructuras de datos (codificación asignada a una variable de entrada o salida).
- D-EJECUTA. Se encarga de decrementar la pila de ejecuciones e inicializarla.
- I-EJECUTA. Se encarga de incrementar la pila de ejecuciones u asignarle el valor tras haber realizado un cálculo.
- POP2, se encarga de invocar 2 veces a la función POPP (necesaria en

determinadas ocasiones.

Ver código módulo INTÉRPRETE en el Anexo I y II, apartado 2.4.

7.5 Módulo de Funciones Adicionales

Este módulo como hemos comentado con anterioridad se ha desarrollado para poder contemplar operaciones futuras no planteadas de origen. No obstante la función LOG se ha incluido en este módulo para que sirva de ejemplo, de cómo implementar nuevas funciones.

Por tanto este módulo sólo contiene dos funciones. A continuación se detallan:

- FUNCIONES. Este módulo contiene un swtich case para poder incluir fácilmente nuevas instrucciones. Dentro del mismo se hará la invocación a la función que tenga que realizar el cálculo. Para las funciones almacenadas en este módulo hay que realizar dos llamadas: la primera nos servirá para poder devolver, en función de la expresión que queremos ejecutar, el número de parámetros de entrada y de salida que contiene; para que en una segunda y posterior llamada se pueda realizar el cálculo. El módulo intérprete en función del número de parámetros, maneja las pilas y estructuras para desapilar y apilar la información.
- LOG. Única función del programa FUNCIONES, contiene la implementación de la instrucción LOG.

Ver código módulo FUNCIONES en el Anexo I y II, apartado 2.5.

7.6 Módulo de Actualización de Parámetros BD-Host

Este módulo ya se ha explicado bastante en detalle en el apartado Módulo Actualización de Parámetros, incluido en el mismo punto un esquema detallado del funcionamiento del mismo.

Las funciones que componen este módulo son:

- HOST. Es el módulo principal, el que contiene toda la lógica y se encarga de invocar al resto de módulos.
- GRABAR-PARAMETROFLUJO, GRABAR-FLUJODECISION, EXTRAER-PARAMETROFLUJO, EXTRAER-ENTORNOMOTOR, COUNT-PARAMETRO-FLUJO, COUNT-FLUJO-DECISION, COUNT-FLUJO-DECISION-A0-E2, COUNT-FD-COD-ENT, DELETE-PARAMETRO-FLUJO, DELETE-FLUJO-DECISION, UPDATE-FLUJO-DECISION y UPDATE-FD-A1-FBAJA. Todas estás son funciones que contiene querys en formato SQL para poder realizar el mantenimiento. Algunas de las mismas aunque no son invocadas desde el programa principal se mantienen para facilitar un posible mantenimiento.

- SQL-VARIABLES. Esta función se encarga de mover el contenido de las variables que se pasan por parámetro a las que requieren las consultas de SQL, que tienen un formato diferente y es necesario para que funcione la asignación de parámetros.
- CONECTA-BBDD y DISCONNECT-BBDD son las funciones que a través de parámetros EXEC SQL establecen o no la conexión con la BBDD.

Ver código módulo ACTPAR en el Anexo I y II, apartado 2.6.

7.7 Módulo de Ejecución desde HOST-BD

El módulo principal de este programa es el módulo HOST. Se encuentra explicado el funcionamiento en el apartado Módulo de entrada HOST

- HOST. Este módulo se encarga entre otras de verificar que el código de flujo y versión permanezcan inalterables durante las diferentes ejecuciones. En caso de que se detecte alguna modificación porque hayan variado los datos pasados por parámetro, se encargará de conectarse a la BBDD, extraer el parámetroflujo correspondiente y cerrar la conexión para posteriormente preprocesar la información tal y como la requiere el módulo COBOL ANADAT (DATOSC). Una vez acabada la ejecución se encarga de trasladar la información obtenida a los parámetros para poder devolver el resultado de la ejecución.
- DATOSC. Esta función transforma los datos obtenidos por parámetros de las cabeceras y variables a la estructura de datos que requiere el módulo ANADAT. Esto tanto se realiza para las cabeceras de entrada como de salida.
- DATOSOU. Se encarga de volcar ciertas variables relevantes de la ejecución como son el código de flujo versión, niveles, número de bloques y cantidad de registros de salida a una estructura que no se borrará al volver a invocar al programa INPUTHOST, de forma que podremos saber cuáles son los valores que existían en una posterior ejecución.
- FLUJO. Esta función es la que invoca al módulo ANADAT, pasándole toda la información que requiere por parámetro.
- EXTRAER-PARAMETROFLUJO. Esta función es la encargada de devolver los parámetros necesarios para poder realizar una ejecución (variables, tira de instrucciones, cabeceras). Esto se hace en función de los parámetros que se le pasan: código, versión, estado, activo y entidad
- DBCONEC establece la conexión con la BD.
- DBCLOSE cierra la conexión con la BD.

Ver código módulo INPUTHOST en el Anexo I y II, apartado 2.7.

8. Conclusiones

En mi etapa como consultor aprendí el manejo de la programación en leguaje COBOL, pero normalmente sólo me había dedicado a modificar trozos de código o funciones muy especificas, por lo que se podría decir que me limitaba a usar 4 o 5 funciones específicas de la totalidad del lenguaje.

Para la realización del proyecto he tenido que estudiar diferentes manuales de referencia de COBOL y documentación muy técnica para ver si las funciones que empleaba eran totalmente compatibles entre el entorno PC y HOST. Cabe recordar que COBOL dispone además de una amplia lista de compiladores y no siempre las funciones empleadas son 100% compatibles entre las diferentes versiones. Esto ocurre porque cada casa se ha dedicado a intentar subsanar las carencias que veía más importantes. Otra razón de estudio de la documentación técnica ha sido para poder llevar a cabo el Tunning de la herramienta, para que ésta tuviese el mayor rendimiento posible. Esto se consigue en parte, gracias al uso de determinados tipos de datos, instrucciones de inicialización de estructuras, etc. También otra fuente empleada ha sido internet, en la cual he encontrado ejemplos y detalle de funciones.

Buena parte del tiempo del proyecto se ha dedicado a la investigación, ya no solo por el lenguaje de programación, sino por el mero hecho que se ha tenido que ver y probar programas de similares características. En determinadas ocasiones, algunas funciones existentes en otros lenguajes y no en COBOL han dificultado la programación, ya que se han tenido que implementar dichas funciones en muchas ocasiones a base de ingenio.

Ha sido un reto poder crear este motor ya que tiene una gran complejidad, debido a que en un primer lugar se ha tenido que crear un lenguaje, y luego se ha tenido que generar un código que lo interpretase. Al fin y al cabo escribir las instrucciones, una vez que se tienen claras las diferentes opciones que se pretenden abordar, puede resultar sencillo, lo más complicado ha sido principalmente el uso de la recursividad para así poder gestionar que la herramienta no tuviese limitaciones a la hora de implementar reglas o bucles ya que como se vio, los programas de la competencia solo permiten un número finito y muy pequeño de instrucciones anidadas. El control de las estructuras de datos, tampoco resulto trivial, ya que se manejan muchas estructuras y de muy variadas características. También una cosa que ha resultado algo tediosa ha consistido en el manejo de los errores, todas las funciones o métodos que podrían ocasionarlos, se han tenido que controlar y se han tenido que implementar las funciones teniendo en cuenta los errores, de forma que estos se hereden de una función a otra, para finalmente poderlos devolver y así tener claramente acotado dónde se han producido.

El proyecto se empezó a abordar buscando cubrir una serie de funcionalidades básicas (principalmente replicar las funcionalidades de los productos de la competencia). A medida que se fue viendo que se lograban cubrir estas funcionalidades, se decidió abordar nuevas. Por ello, a medida que se ha ido implementando el proyecto, se ha tenido que ir modificando para hacerlo más eficiente y sencillo de usar.

A pesar de siempre buscar la sencillez e intentar generar un código claro de entender, se tuvo que realizar una tarea de Tunning final, que a su vez sirvió para aún más simplificar el código. Se han intentando incluir en funciones o métodos todos aquellos trozos de código repetido, aunque resultasen de lo más sencillos.

La programación del código COBOL está acotado por el número de columnas que se pueden programar, otro punto que también en parte dificulta la identación del código para que éste sea claro. Por ello, cuando se programa en COBOL, la programación debe ser de lo más estructura posible.

Por otra parte, realizar no sólo un módulo para entorno PC sino también para HOST es otro punto a favor a valorar, ya que parte del funcionamiento lo alteramos, ya que se ha puesto en juego la programación incluyendo la comunicación con la BBDD y teniendo que retocar alguno de los métodos iniciales ya que el entorno HOST no los soportaba. Cosa que también implicó el estudio de cómo se conjuga el lenguaje COBOL con SQL, que aunque parezca sencillo, tiene algún que otro truco.

Finalmente, creo que el objetivo de lograr la independencia entre los usuarios y los equipos de DyD se ha conseguido, ya que con este motor, y construyendo posteriormente una interface gráfica que genere el código que requiere, el usuario va a poder realizar cualquier tipo de flujo que necesite. Adicionalmente el usuario, va a poder estar seguro de que los cálculos que se realicen dentro del motor, van a ser totalmente fiables, ya que es una de las características que aporta el lenguaje COBOL y que no esta presente en otros programas existentes en el mercado.

9. Trabajos Futuros

Debido al tiempo que se ha empleado en la construcción de esta herramienta, que nos ha permitido en gran medida conseguir la independencia entre el usuario y los departamentos técnicos, no ha sido posible la elaboración de un programa que genere el pseudocódigo que requiere. Por ello, como segunda fase se propone la construcción de un compilador o parser, que se encargue, partiendo de un pseudocódigo, generar las instrucciones que requiere el motor para así poder ser evaluadas.

Este compilador o parser tendría que poder verificar en un primer lugar que las tiras de instrucciones generadas estuviesen compuestas de forma correcta, que las asignaciones entre los diferentes tipos de datos se controlase y advirtieran al usuario que la acción que esta realizando esta o no permitida. También tendría que proporcionar métodos para que la construcción de determinados elementos complejos como las reglas o los bucles resultasen lo más sencillos posibles (vía asistentes, etc...).

Adicionalmente al compilador, y a través de una interface gráfica se tendría que poder gestionar la generación de los flujos y que estos acabasen convirtiéndose de forma automática en los elementos o funcionalidades que permitan al motor realizar el linkando de las etiquetas y así como las directivas de saltos, etc.

Otra funcionalidad adicional podría consistir en desarrollar asistentes para permitir de forma ágil construir los ficheros que el motor necesita, no solamente la tira de instrucciones, sino el fichero de cabeceras de entrada y de salida, así como también el de variables.

Incluso se podría llegar a desarrollar un programa que preprocesase los datos con los cuales fuese a ser evaluada la información, para así poder obtener los catálogos de los cuales dispone de cada una de las variables y así, a través del editor, simplificar al usuario la generación de los flujos.

10. Pruebas

Para la realización de las pruebas se han generado diversos juegos de pruebas, empezando en un primer lugar por las versiones más sencillas que pretendían probar funcionalidades básicas de la herramienta, hasta los últimos juegos de pruebas que incorporan múltiples reglas y toda la complejidad que se ha podido desarrollar.

Juego de Pruebas 1:

Este primer juego de pruebas pretende probar la complejidad de tener reglas incorporadas dentro de otras reglas y al mismo tiempo dentro de las mismas puedan existir acciones para el caso de que se cumplan las condiciones como también si no se cumplen.

Pseudocódigo:

```
V2001= 25
V3001 = 0
V3003 = 0
V3004 = 0
SI V1001 = 1 THEN
        V3001 = 100 +  1
ENDIF
SI V3001 = 101 Y V1002 = 2 THEN
        V3002 = (V3001 * V1002) + 2
ELSE
        V3002 = (200 * V1002) + 3
ENDIF
SI V1003 = V1004 THEN
        SI V1005 = V1006 THEN
                V3003 = V3002 * (V1002 / 2)
        ELSE
                V3003 =  V3002 * (V1004 /2)
        ENDIF
ELSE
        SI V1005 <> V1006 THEN
                SI V1001 > V1002 THEN
                        SIV1001 > V1003 THEN
                                V3004 = V1001 *1000
                        ELSE
                                V3004 = (V1001 * 1000) / 2
                        ENDIF
                ELSE
                        V3004 =  (V1001 * 1000) / 3
                ENDIF
        ELSE
                V3004 = (V1001 * 1000) / 4
        ENDIF
ENDIF
V3005= (V3001 + V3002 + V3003 + V3004) / V2001
```

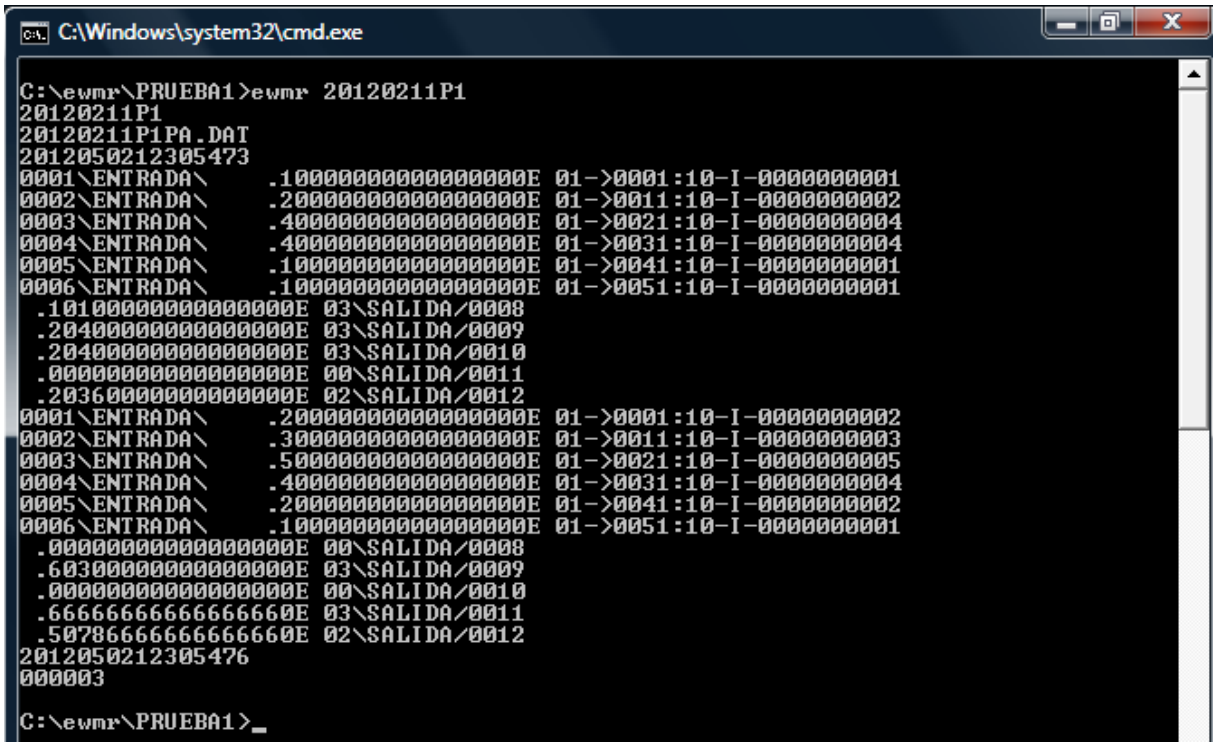
Flujo en versión intérprete:

```
Q001
V1006(,25#Fv
V1007(,0#Fv
V1009(,0#Fv
V1010(,0#Fv
R1,1000#(,1#7,E#F
        A1007(,100#(,1#7,8#Fa
```

```
r
R1,1007#(,101#7,E#1,1001#(,2#7,E#7,H#F
        A10081,1007#1,1001#7,=#(,2#7,8#Fa
e
        A1008(,200#1,1001#7,=#(,3#7,8#Fa
r
R1,1002#1,1003#7,E#F
        R1,1004#1,1005#7,E#F
                A10091,1008#1,1001#(,2#7,>#7,=#Fa
                e
                A10091,1003#1,1001#(,2#7,>#7,=#Fa
        r
e
        R1,1004#1,1005#7,F#F
                R1,1000#1,1001#7,A#F
                        R1,1000#1,1002#7,A#F
                                A10101,1000#(,1000#7,=#Fa
                        e
                                A10101,1000#(,1000#7,=#(,2#7,>#Fa
                        r
                e
                        A10101,1000#(,1000#7,=#(,3#7,>#Fa
                r
        e
                        A10101,1000#(,1000#7,=#(,4#7,>#Fa
        r
r
V10111,1007#1,1008#7,8#1,1009#7,8#1,1010#7,8#1,1006#7,>#Fv
W
```

Resultados (entrada y salida):

	V1001	V1002	V1003	V1004	V1005	V1006	V3001	V3002	V3003	V3004	V3005
Reg1	1	2	4	4	1	1	101	204	204	0	20.36
Reg2	2	3	5	4	2	1	0	603	0	666.6666	50.7866



A continuación se adjuntan los ficheros necesarios para realizar la prueba.


20120211P1DI.DAT


20120211P1DO.DAT


20120211P1ER.DAT


20120211P1FI.SEM


20120211P1PA.DAT

Una de las grandes ventajas que se puede apreciar en este juego de pruebas, consiste en que si se descomponen los tokens, se puede ver que no hace falta alterar el orden de las operaciones, se pican tal cual aparecen en el pseudocódigo (teniendo en cuenta que el operador se pone a continuación de 2 operandos). Cosa que hace que la generación de las instrucciones que requiere la herramienta sea más sencillas de construir.

Juego de Pruebas 2:

Este segundo juego de pruebas tiene como principal objetivo realizar pruebas sobre la funcionalidad de bucle, demostrando que se pueden incluir tanto otras reglas o bucles dentro del mismo.

Pseudocódigo:

```
V2001 = 1
V3002 = 0
V3003 = 0
MIENTRAS (10000, V2001 < V1001) hacer
        V3002 = V3002 + V2001 * V1002
        SI V3002 < V1003 ENTONCES
                V3002 =V1003 + 1
        ELSE
                V3002 = V3002 * 1,5
        FINSI
        MIENTRAS (50,V3003 < 5)
                V3004 = V3003 * 5
                V3003 = V3003 + 1
        FINMIENTRAS
        V2001 = v2001 + 1
FIN MIENTRAS
V3001 = V2001
```

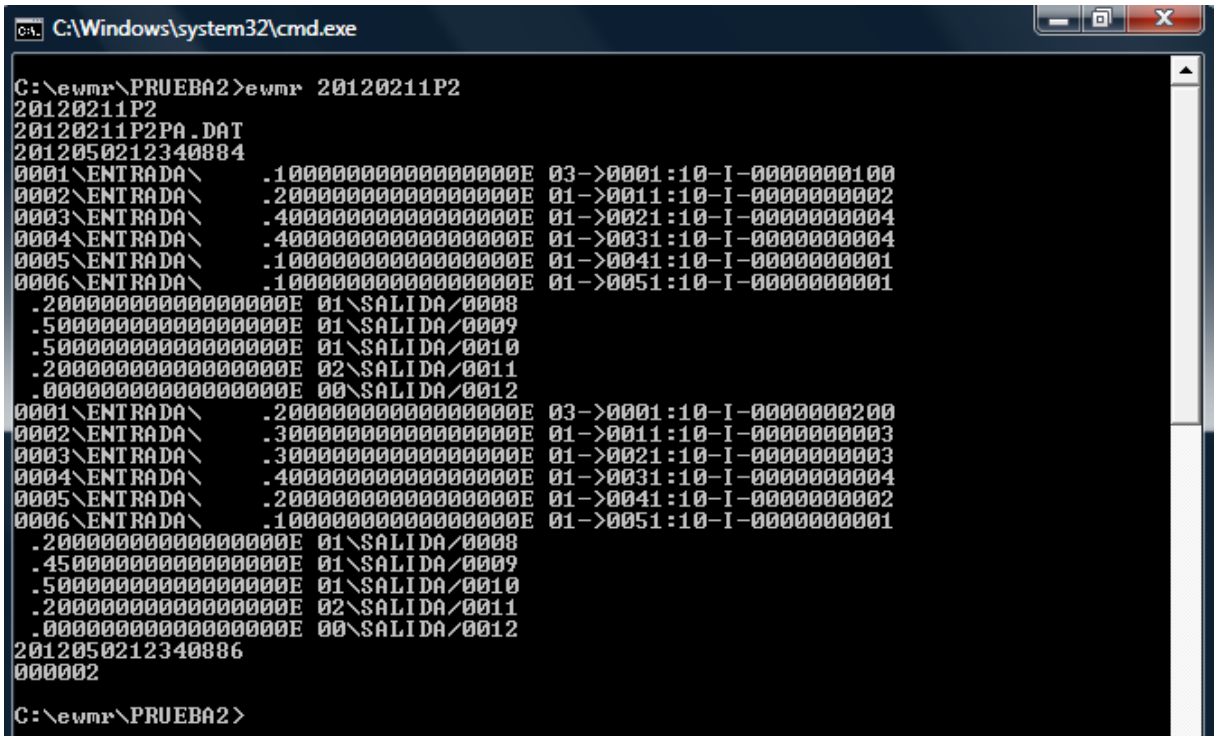
Flujo en versión intérprete:

```
Q001
V1006(,1#Fv
V1008(,0#Fv
V1009(,0#Fv
B100001,1006#1,1000#7,D#F
        10081,1008#1,1006#1,1001#7,=#7,8#Fa
        R1,1008#1,1002#7,C#F
                A10081,1002#(,1#7,8#Fa
        e
                A10081,1008#(,1.5#7,=#Fa
        r
        B000501,1009#(,5#7,C#F
                A10101,1009#(,5#7,=#Fa
                A10091,1009#(,1#7,8#Fa
        b
        A10061,1006#(,1#7,8#Fa
b
```

V10071,1006#Fv
W

Resultados (entrada y salida):

	V1001	V1002	V1003	V1004	V1005	V1006	V3001	V3002	V3003	V3004	V3005
Reg1	100	2	4	4	1	1	2	5	5	20	0
Reg2	200	3	3	4	2	1	2	4,5	5	20	0



A continuación se adjuntan los ficheros necesarios para realizar la prueba.


20120211P2DO.DAT


20120211P2ER.DAT


20120211P2FI.SEM


20120211P2PA.DAT


20120211P2DI.DAT

Juego de Pruebas 3:

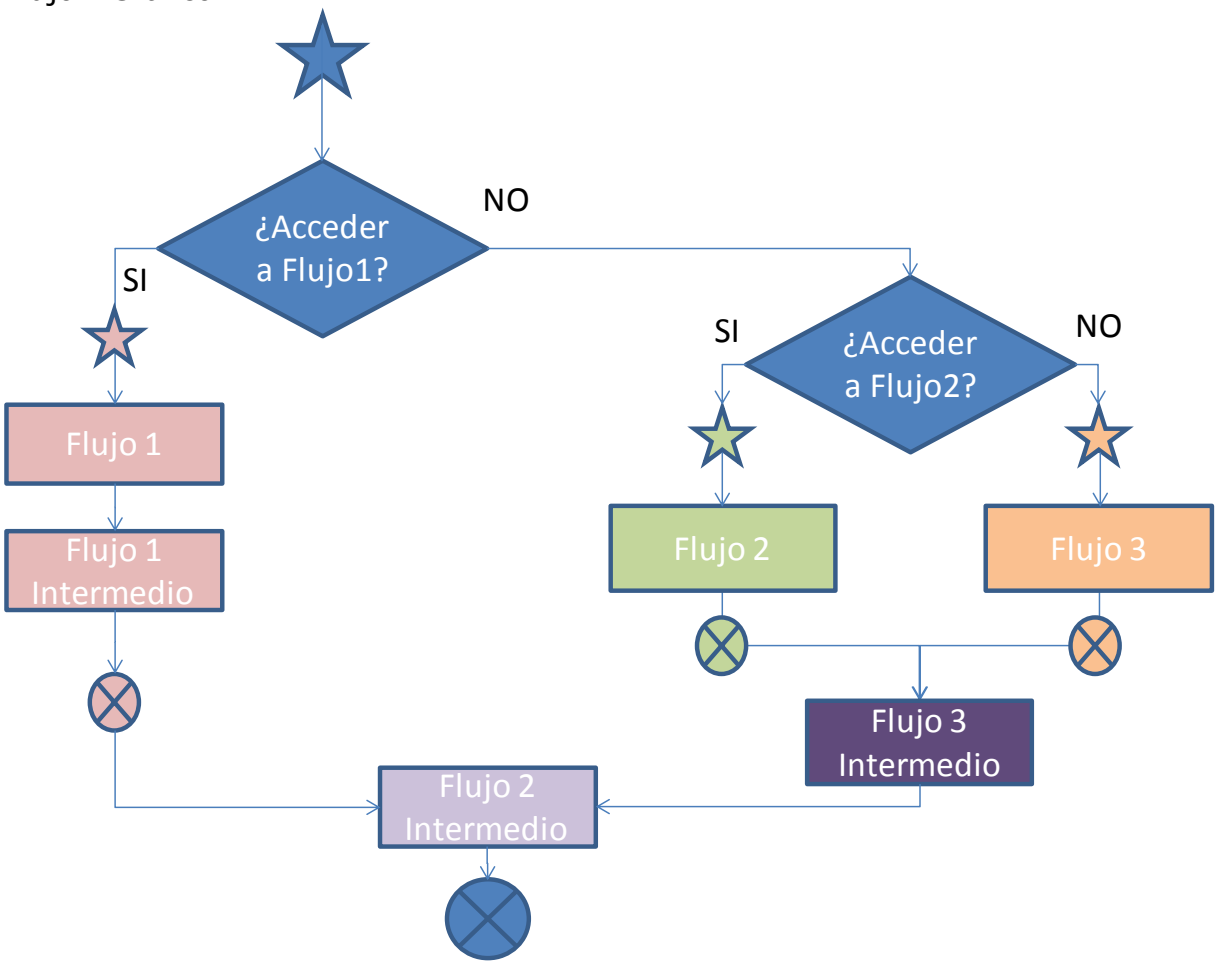
Este juego de pruebas pretende demostrar como se pueden incorporar sub-flujos dentro de un flujo principal y como se pueden llevar a cabo las diferentes llamadas entre los mismos, para poder llegar a soportar el esquema o gráfico que se muestra dentro del ejemplo.

Pseudocódigo

```
SI V1001 = 1 ENTONCES
       IR_A (FLUJO1)
```

```
ELSE
        SI V1001 = 2 ENTONCES
                IR_A (FLUJO2)
        ESLE
                IR_A(FLUJO3)
        ENDIF
FINSI

INI-FLUJO1
        VAR3001= VAR1002 * VAR1003
        VAR3006 =”FLUJO1”
        SALTO(FLUJO-INTERMEDIO1)

FIN-FLUJO1

INI-FLUJO2
        VAR3002 = VAR1003 * VAR1004
        VAR3006=”FLUJO2”
        SALTAR(FLUJOINTERMEDIO3)
FIN-FLUJO2

INI-FLUJO3
        VAR3003 = VAR1004 * VAR1005
        VAR3006=”FLUJO3”
        SALTAR(FLUJOINTERMEDIO3)
FIN-FLUJO3

INI-FLUJO-INTERMEDIO1
        VAR3004 = VAR3001 + 1
        VAR3005 = CONCATENAR(VAR3005,”-INT1”)
        SALTAR(FLUJOINTERMEDIO2)
FIN-FLUJO-INTERMEDIO1

INI-FLUJO-INTERMEDIO2
        VAR3004 = VAR3004 + 1
        VAR3005 = CONCATENAR(VAR3005,”-INT2”)
        SALTAR(FLUJOFINAL)
FIN-FLUJO-INTERMEDIO1

INI-FLUJO-INTERMEDIO3
        SI V1001 = 2 ENTONCES
                VAR3004 = VAR3002 + 1
        ELSE
                VAR3004 = VAR3003 + 1
        ENDIF
        VAR3005 = CONCATENAR(VAR3006,”-INT3”
        SALTAR(FLUJOINTERMEDIO2)
FIN-FLUJO-INTERMEDIO3

INI-FLUJOFINAL
        FIN
FIN-FLUJOFINAL
```

Flujo - Gráfico



Flujo en versión intérprete:

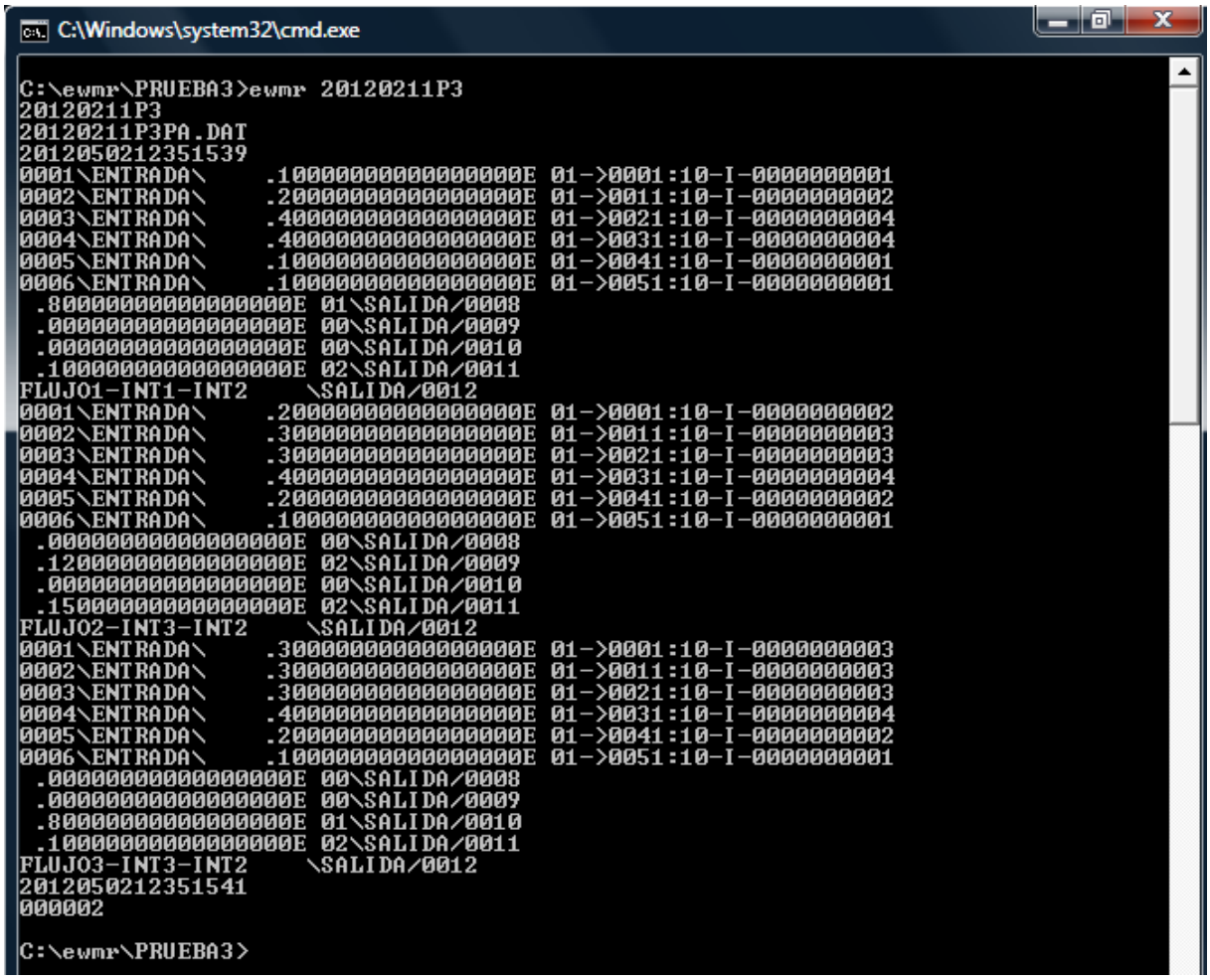
```
Q001
R1,1000#(,1#7,E#F
        S002
e
        R1,1000#(,2#7,E#F
                S003
        e
                S004
        r
r
Q002
        V10071,1001#1,1002#7,=#Fv
        V1011*,&FLUJO1&#Fv
        S005
Q003
        V10081,1002#1,1003#7,=#Fv
        V1011*,&FLUJO2&#Fv
        S006
Q004
        V10091,1003#1,1004#7,=#Fv
        V1011*,&FLUJO3&#Fv
        S006
Q005
        V10101,1007#(,1#7,8#Fv
        V10115,1011#*,&-INT1&#7,M#Fv
        S007
Q007
        V10101,1010#(,1#7,8#Fv
        V10115,1011#*,&-INT2&#7,M#Fv
        S008
Q006
        R1,1000#(,2#7,E#F
                A10101,1008#(,2#7,8#Fa
        e
                A10101,1009#(,1#7,8#Fa
        r
        V10115,1011#*,&-INT3&#7,M#Fv
        S007
Q008
W
```

ENTRADA

	V1001	V1002	V1003	V1004	V1005	V1006
Reg1	1	2	4	4	1	1
Reg2	2	3	3	4	2	1
Reg3	3	3	3	4	2	1

SALIDA

	V3001	V3002	V3003	V3004	V3005
Reg1	8	0	0	10	FLUJO1-INT1-INT2
Reg2	0	12	0	15	FLUJO2-INT3-INT2
Reg3	0	0	8	10	FLUJO3-INT3-INT2



A continuación se adjuntan los ficheros necesarios para realizar la prueba.


20120211P3PA.DAT


20120211P3DI.DAT


20120211P3DO.DAT


20120211P3ER.DAT


20120211P3FI.SEM

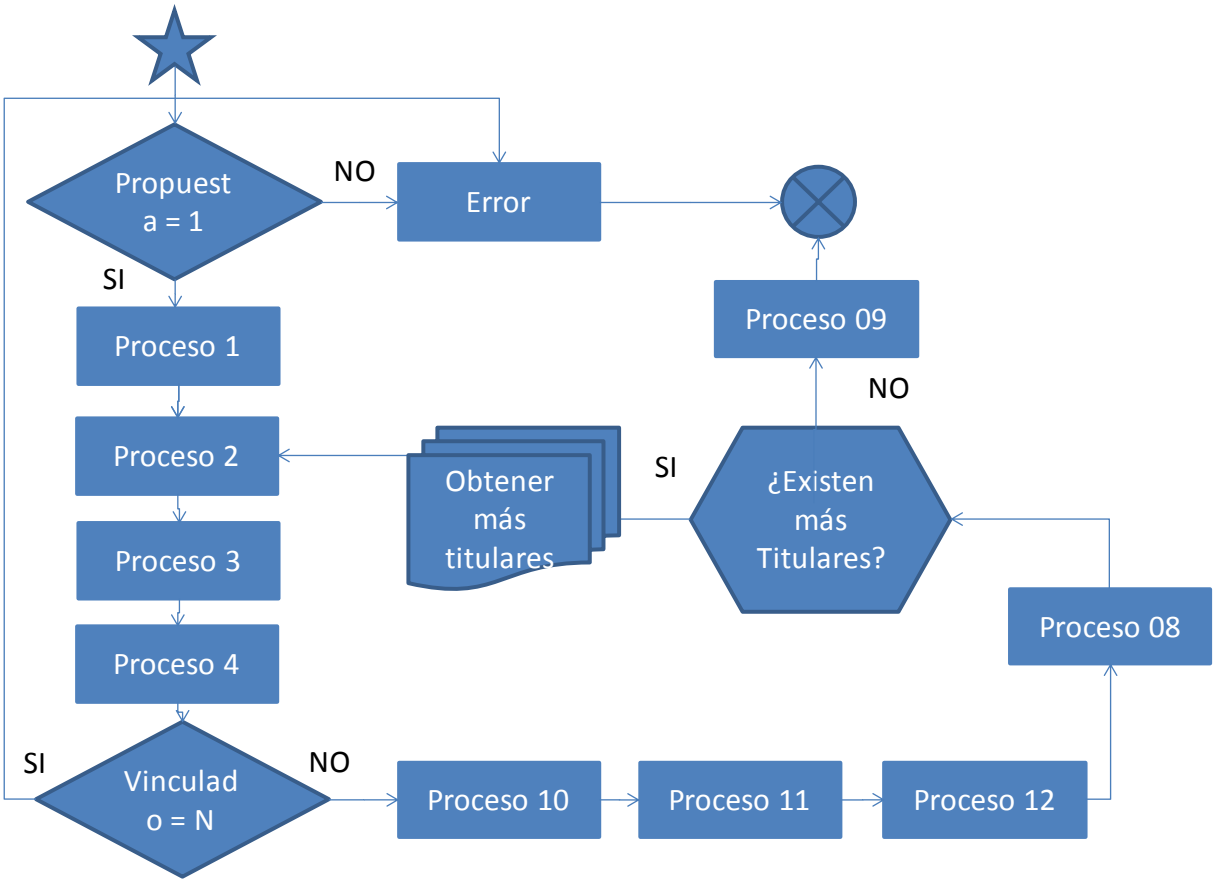
Juego de Pruebas 4:

Este juego de pruebas es de los más amplios y complicados de los que se han realizado, llevando a la práctica la mayoría de las funcionalidades que se han implementado. Este ejemplo, pretende contemplar un flujo completo correspondiente a la contratación de un producto bancario, en concreto de un producto de consumo. En él, se muestran las diferentes fases para dictaminar si a un interviniente se le concede un préstamo en función de su vinculación, perfil, su balance de caja, etc.

A continuación se explican los diferentes módulos:
INI000 – Módulo para tratar las variables con flag de tipo OTRO, en función de los NO-INFO
INI001 – Inicialización de variables intermedias
INI002 – Tratamiento de variables de perfil e inicialización de variables de salida.
INI003 – Cálculo del Balance de caja, en función de la provincia, ingresos, número de componentes de la unidad familiar, etc…
INI004 – Análisis de vinculación
INI005 – Cálculo de ratios; cobertura, cuota, financiación y análisis de profesiones o destino del bien a adquirir. No se evalúa.
INI006 – Cálculo de las variables de operación que no intervienen en el modelo con sus ponderaciones. No se evalúa.
INI007 – Cálculo total, probabilidad de mora y dictámenes de los solicitantes Hipotecas. No se evalúa.
INI008 – Cálculo final de dictámenes de perfil incluyendo reglas de discriminación. No se evalúa.
INI009 – Conclusión definitiva y ponderación de los diferentes titulares.
INI010 – Cálculos de categorización de las variables del modelo.
INI011 - Cálculo de las variables de operación que intervienen en el modelo con sus ponderaciones.
INI012 – Cálculo total, probabilidad de mora y dictámenes de los solicitantes Consumo

Al tratarse de un flujo tan completo y extenso, en el Anexo Pruebas, se puede encontrar el detalle del pseudocódigo empleado, apartado 1.1

A continuación se muestra un esquema que muestra el flujo descrito anteriormente.



En el Anexo Pruebas apartado 1.2, se puede encontrar el detalle del código generado para poder contemplar el pseudocódigo anteriormente descrito.

Debido al gran volumen de variables empleadas, no se adjunta tabla con datos de entrada y salida. Los resultados, al igual que en las otras pruebas se pueden ver en el fichero de salida (acabado en DO.DAT). No obstante se adjunta fichero SALIDA.TXT para que se vean las variables de entrada y de salida con sus datos tras la evaluación.

SALIDA.TXT

A continuación se adjuntan los ficheros necesarios para realizar la prueba.

20120211P4DO.DAT 20120211P4ER.DAT 20120211P4FI.SEM 20120211P4PA.DAT

20120211P4DI.DAT

11. Instalación

La instalación de este programa dependerá principalmente del entorno en el cual necesitemos que sea ejecutado (PC o HOST), ya que se tienen que llevar a cabo una serie de instrucciones especiales para poder realizar una primera ejecución del programa.

Como se ha visto anteriormente, los módulos de programación se dividen principalmente en dos grandes bloques: los módulos puros de COBOL (con extensión .CBL) y los módulos que contiene programación COBOL + código SQL (con extensión SQB). Estos módulos para que sean comprendidos por la máquina o el entorno en el cual van a ser ejecutados deben ser traducidos, normalmente a "lenguaje máquina" o a un código intermedio (bytecode). Este proceso se conoce por el nombre de compilación.

A parte de permitirnos realizar esta traducción, pasando de un lenguaje de alto nivel a otro de nivel inferior, el proceso de compilación tiene otras finalidades como son:

- análisis léxico, que consiste en la lectura del código fuente agrupando la información en tokens (componentes léxicos) que son caracteres que tiene un significado. Se eliminarán todos los espacios y líneas en blanco, así como los comentarios y demás código no relevante correspondiente al programa fuente. Por otra parte una función muy importante consiste en la comprobación que los símbolos del lenguaje (operadores, palabras clave, instrucciones) se hayan escrito de forma correcta.

- análisis sintáctico, en esta fase los caracteres o componentes léxicos se agrupan jerárquicamente en frases gramaticales que el compilador utiliza para sintetizar la salida. Sobre el resultado del análisis léxico se comprueba si es sintácticamente correcto.

- análisis semántico, se encarga de la revisión del programa fuente para tratar de encontrar errores semánticos y reúne la información sobre los tipos para la fase posterior de generación de código. En este paso se realiza la verificación de tipos, comprobando que cada operador, tenga operandos permitidos por la especificación del lenguaje fuente.

- fase de síntesis. En este punto es cuando se genera el código objeto equivalente al programa fuente. Llegados a este punto, tras la generación, es cuando podemos tener la certeza que el programa esta libre de errores de análisis, lo cual nos indicará que el programa se ejecutará de forma correcta.

- generación de código intermedio. Se corresponde con una representación intermedia explícita del programa fuente, la cual tiene dos propiedades importantes: la simplicidad de producirla y la fácil traducción al programa objeto.

- optimización de código. Ésta es la última fase del proceso, y pretende

mejorar el código intermedio de forma que resulte un código máquina más rápido de ejecutar.

Este proceso de compilación es un paso necesario para poder llevar a cabo la instalación del producto, sobre todo al permitirse la ejecución en diferentes entornos, ya que como se ha mencionado el ejecutable variará en función de la máquina o el entorno. A continuación vamos a ver los diferentes métodos de compilación, así como los pasos necesarios para poder llevar a cabo la ejecución sobre el entorno PC, ya que la instalación sobre entorno HOST es exclusiva en función de la versión de SO que disponga el Mainframe.

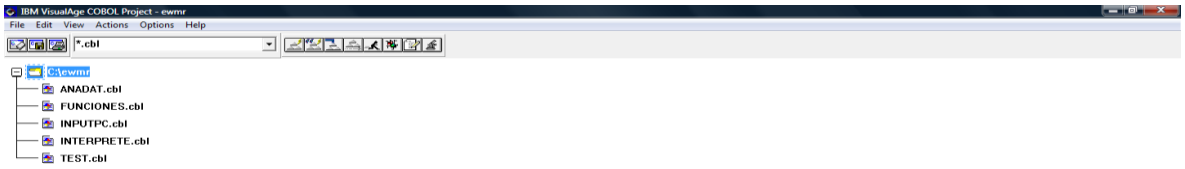
11.1 Instalación en entorno PC

Como se ha comentado anteriormente en el documento, se ha realizado un módulo de programación COBOL (TEST) para poder gestionar de forma más sencilla la realización de las pruebas, ya que con sólo recuperar un parámetro se iniciará la ejecución, que se corresponderá con la cabecera del fichero a usar como input de datos para poder ser invocado. Este módulo, al tratarse del módulo que realiza las llamadas al resto de los mismos, se convertirá en el módulo principal que se invoque para realizar el compilado de los módulos COBOL para el entorno PC.

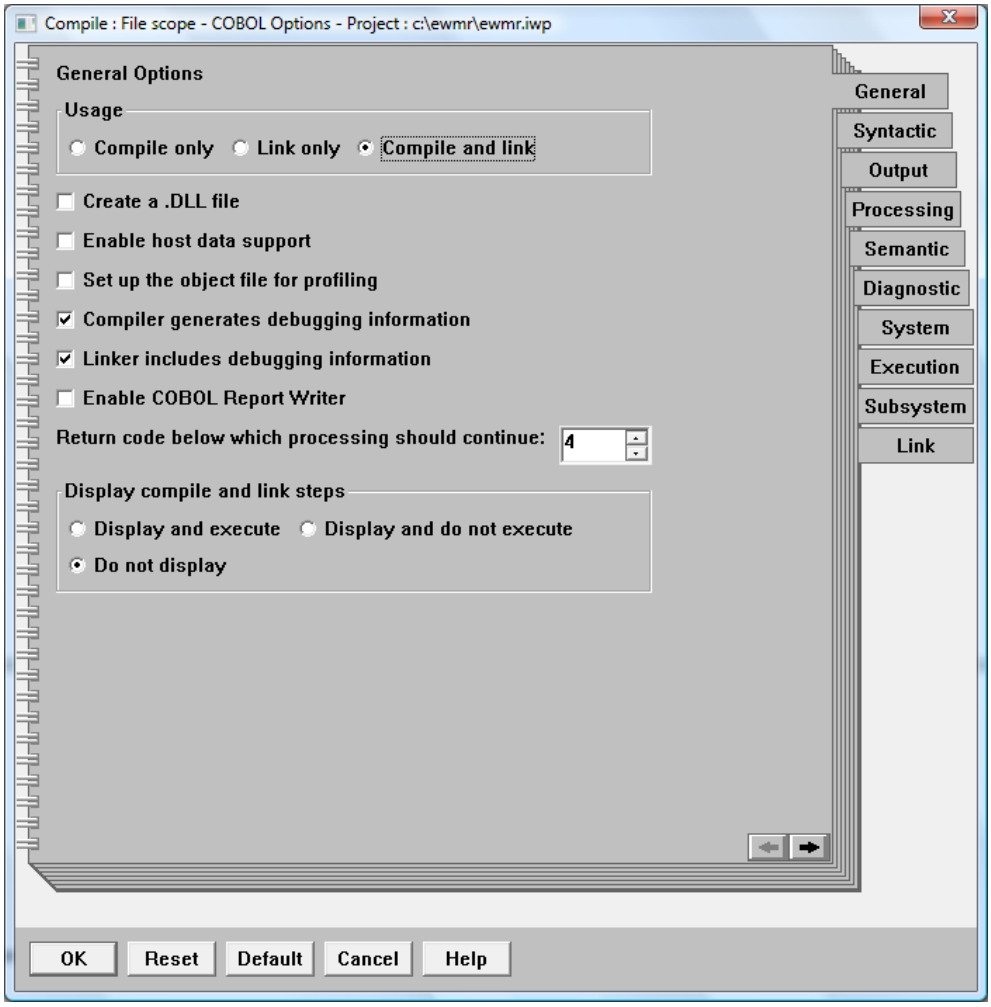
La compilación de los módulos se podrá realizar usando un asistente o programa de compilación en el cual indiquemos la configuración de compilado, o bien usando procedimientos batch que nos permitan realizar la compilación.

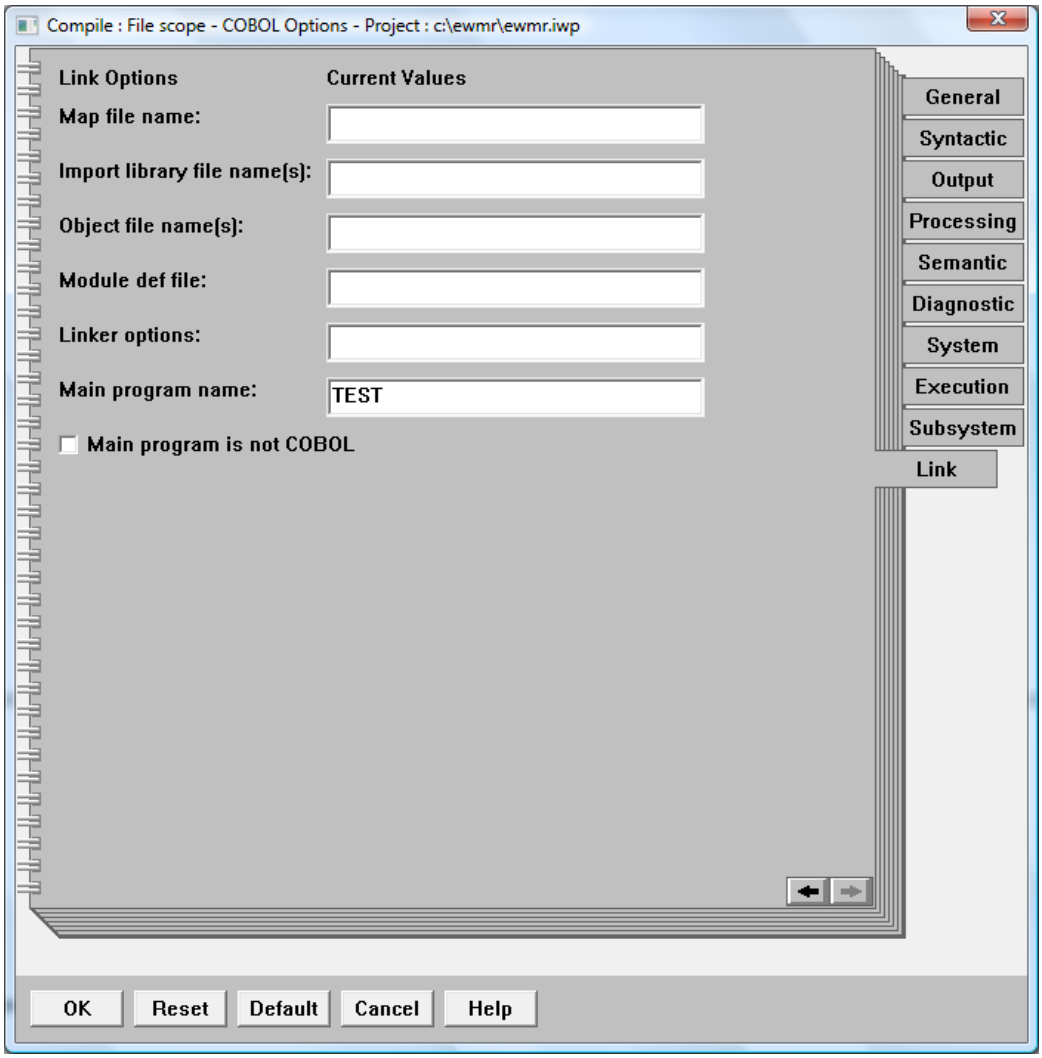
En una fase inicial de proyecto el compilado se realizaba vía asistente. Es más sencilla la generación ya que con indicarle el nombre del programa principal, el programa se encarga de generar la salida, o lo que es lo mismo el fichero ejecutable.

El programa empleado, para este primera fase de desarrollo, fue IBM Visual Age COBOL, el cual tras crear un nuevo proyecto la visión principal es la siguiente:

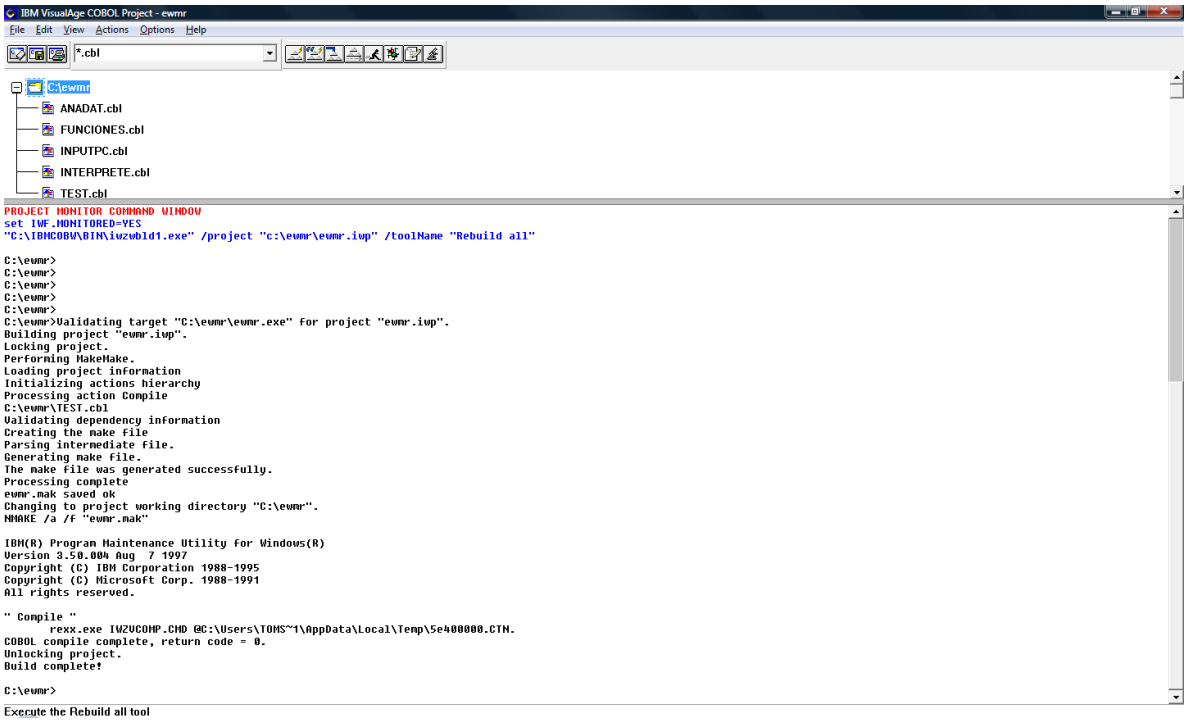


En la siguientes imágenes se muestran los parámetros que son necesarios modificar partiendo de la configuración básica del proyecto.

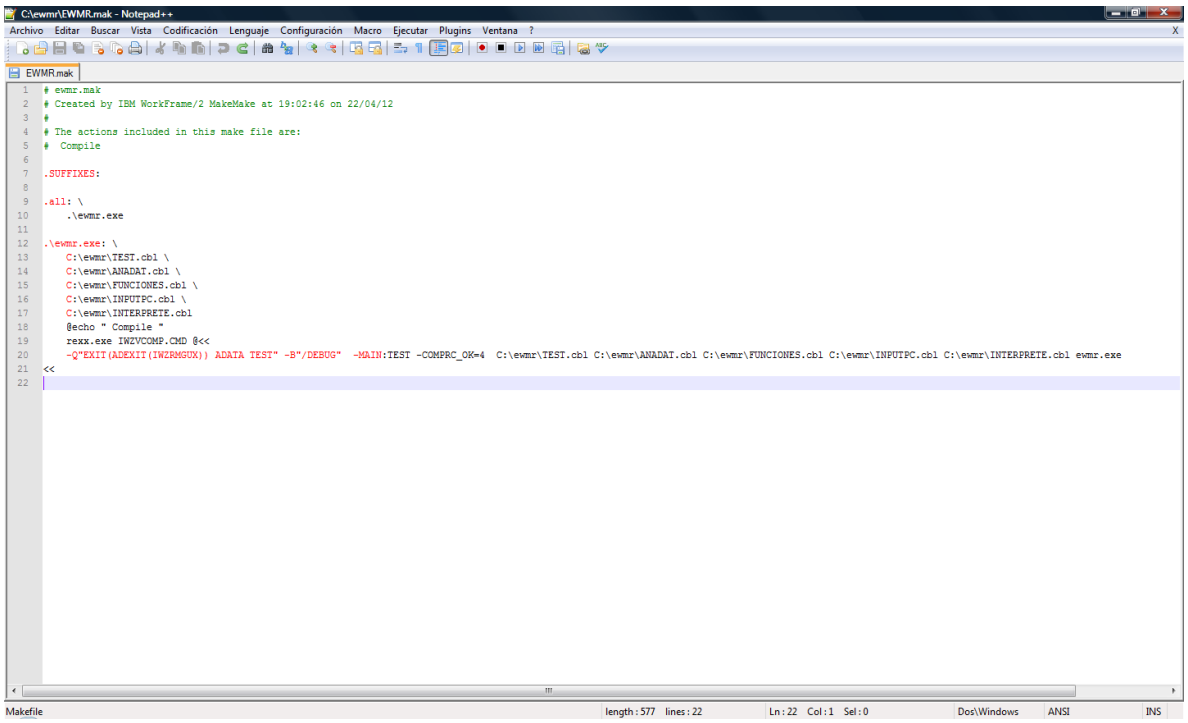




En la imagen que se muestra a continuación se puede ver el resultado de una compilación real del programa. Siempre y cuando se nos retorne por pantalla un "return code = 0" nos indicará que la compilación se ha llevado a cabo con éxito (sin ningún error).

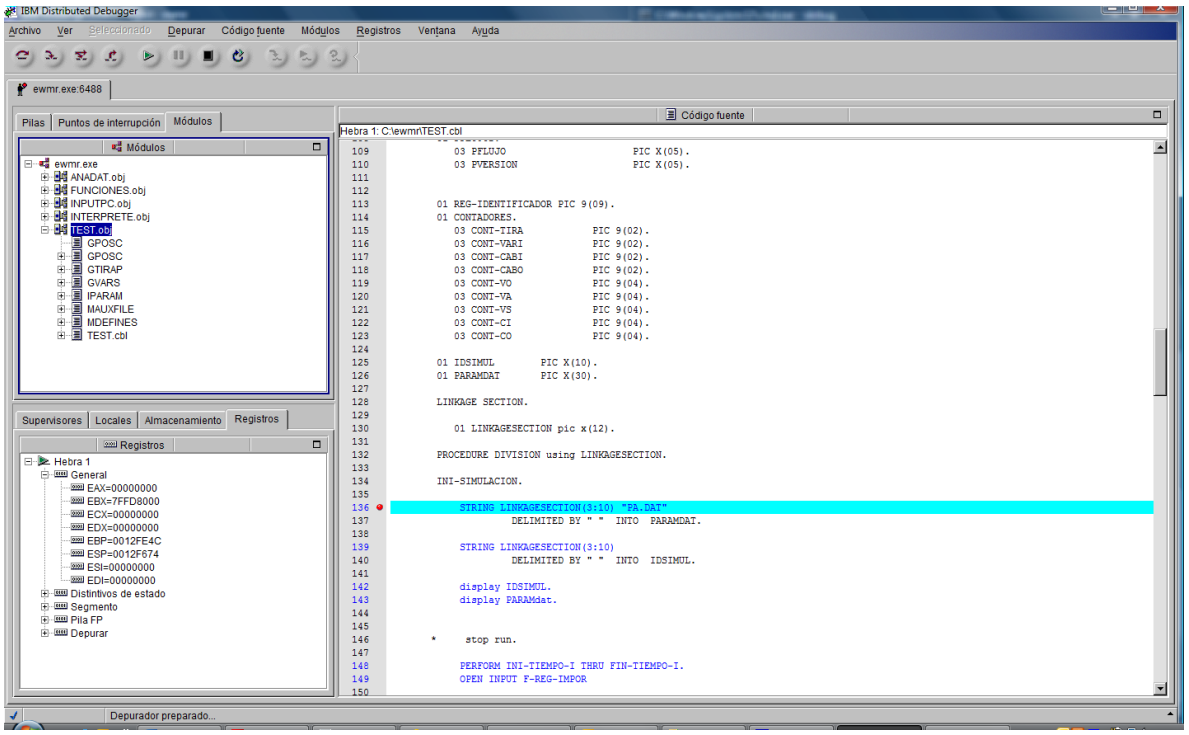


A continuación se puede ver el fichero EWMR.mak que contiene los parámetros para poder realizar la compilación:



La principal ventaja que existe al utilizar el programa IBM Visual Age ha sido que nos permite disponer de las opciones de DEBUG usando un programa que viene embebido dentro del programa VisualAge.

A continuación se muestra una imagen correspondiente al programa idebug (depurador o debugger de IBM):



Una vez construidos los módulos principales y se empezaron a implementar los módulos de BBDD, se tuvo que descartar el uso del programa VisualAge, ya que el debugger no permitió la compilación ni depuración incluyendo los módulos de BBDD.

A continuación nos vamos a centrar en la compilación vía procedimiento batch, ya que nos permite ser más independientes y poder realizar un compilado más configurable, indicando incluso si no es necesaria la compilación de módulos de BBDD como son los módulos SQB.

A continuación se muestra el código empleado para llevar a cabo esta compilación y que incluye tanto la compilación del programa usando ficheros, como también el programa conectado a BD invocado desde PC.

MAKE.BAT

```
::************************************************ ::
::                                                 ::
:: COMPILACIÓN EWMR                                ::
::                                                 ::
:: Incluye precompilación de módulos sqb (con embedded-  ::
:: SQL),compilación de módulo cbl y linkado de todo el proyecto.  ::
::                                                 ::
:: Autor: TSR                                      ::
::                                                 ::
::************************************************ ::
```

```
:: BORRADO DE PANTALLA
cls

:: AÑADIMOS A LA VARIABLE DE ENTORNO "COBPATH" LA RUTA
       DONDE COMPILAMOS

set COBPATH=%COBPATH%;D:\EWMR;
set COBPATH=..\;

:: ELIMINAMOS COMPILADOS...
del *.obj
del *.dll
del *.lst
del *.exp
del *.imp
del *.lib
del *.adt
del TEST.exe
del INPUTPC.exe
del *.dll

:: CONEXIÓN A BASE DE DATOS
rem
rem
       ____________________________________________________________
       ____________________________________________________

rem Conectando a base de datos
db2 connect to ewmr user db2admin using db2admin

:: PRECOMPILACIÓN DE FICHEROS SQB
rem
rem
       ____________________________________________________________
       ____________________________________________________

rem Precompilación de ficheros sqb
rem Precompilado de Motor
call prep.bat INPUTHOST
rem Precompilado del modulo de importación de parámetros
call prep.bat ACTPAR

:: DESCONEXIÓN DE LA BASE DE DATOS
rem
rem
       ____________________________________________________________
       ____________________________________________________

rem Desconectando de base de datos
db2 connect reset
```

```
:: COMPILACIÓN DE "ERROR CHECKING FACILITY"
rem
rem
        ____________________________________________________________
        ____________________________________________________

rem Compilación de "error checking facility"
cob2 -qpgmname(mixed) -c -qlib -I%DB2PATH%\include\cobol_a
        checkerr.cbl

:: COMPILACIÓN DE LOS MÓDULOS CBL PROVENIENTES DE
        PRECOMPILACIÓN DE SQB
rem
rem
        ____________________________________________________________
        ____________________________________________________

rem Compilación de módulos cbl provenientes de precompilación de
        sqb
rem     Compilado de Módulos del Motor sqb .....................................
cob2 -qpgmname(mixed) -c -qlib -I%DB2PATH%\include\cobol_a
        INPUTHOST.cbl
rem     Compilado de Módulos del Importador sqb .....................................
cob2 -qpgmname(mixed) -c -qlib -I%DB2PATH%\include\cobol_a
        ACTPAR.CBL

:: COMPILACIÓN DE MÓDULOS CBL (INDEPENDIENTES DE LOS
        MÓDULOS SQB)
rem
rem
        ____________________________________________________________
        ____________________________________________________

rem Compilación de módulos cbl
rem     Compilación de los módulos restantes del Motor
cob2 -c ANADAT.cbl FUNCIONES.cbl INTERPRETE.cbl TEST.cbl
rem     Compilación de los módulos restantes del Importador
cob2 -c INPUTPC.cbl

:: LINKADO DEL PROGRAMA
rem Generación de Librerias (lib) y linkado con generación de dll del
        Motor
ilib /nol /gi:inputhost inputhost.obj
ilib /nol /gi:anadat anadat.obj
ilib /nol /gi:interprete interprete.obj
ilib /nol /gi:funciones funciones.obj
ilink /free /nol /dll db2api.lib inputhost.exp inputhost.obj
        iwzrwin3.obj
ilink /free /nol /dll db2api.lib anadat.exp anadat.obj iwzrwin3.obj
ilink /free /nol /dll db2api.lib interprete.exp interprete.obj
        iwzrwin3.obj
ilink /free /nol /dll db2api.lib funciones.exp funciones.obj
```

```
        iwzrwin3.obj

rem Generación de Librerias (lib) y linkado con generación de dll del
        Importador

ilib /nol /gi:actpar actpar.obj
ilink /free /nol /dll db2api.lib actpar.exp actpar.obj iwzrwin3.obj


rem
        ____________________________________________________________
        ____________________________________________

rem Linkando del programa Principal Generación del EXE del Motor
cob2 -o -g TEST.OBJ checkerr.obj db2api.lib
rem Linkando del programa Principal Generación del EXE del
        Importador
cob2 -o -g INPUTPC.OBJ checkerr.obj db2api.lib


rem Copy stored procedure to the %DB2PATH%\function directory.
rem Substitute the path where DB2 is installed for %DB2PATH%.
rem copy Inputhost.dll %DB2PATH%\function
rem copy anadat.dll %DB2PATH%\function


rem Eliminación de ficheros no necesarios...
del *.obj
del *.lst
del *.lib
del *.adt
del *.exp
rem del *.imp

del actpar.cbl actpar.bnd
del inputhost.cbl inputhost.bnd

@echo on
```

PREP.BAT

```
db2 prep %1.sqb bindfile target ibmcob
db2 bind %1.bnd
```

Una vez que tenemos los objetos ejecutables generados (fichero .EXE) en caso de la compilación vía programa VisualAge o fichero EXE y ficheros DLL (Dynamic Link Library), uno por cada módulo adicional, en el caso de la compilación vía comandos batch, realizaremos los siguientes pasos:

1.- crearemos una carpeta en la ruta donde queramos realizar la ejecución del programa.
2.- copiaremos a dicha carpeta los ficheros generados EXE o EXE y DLLs
3.- copiaremos o bien en la carpeta C:\windows\system32 o en la carpeta de ejecución, dos librerías adicionales que son las que mencionamos a continuación:
- ARZLITE.dll
- IWZRLIB.dll

Estas dlls no son registrables, por lo que copiándolas en el directorio System32 que forma parte de los paths (rutas especificas) usadas por el SO Windows la ejecución funcionará de forma correcta. La ventaja de copiarlas en System32 reside en que si tenemos diferentes programas ejecutables en diferentes directorios, y estos programas requieran de dichas librerías nos permitirá no tener que replicar las dll por cada una de los directorios.

Finalizado este último paso ya se da por finalizada la instalación en el entorno PC.

12. Bibliografía

Para la realización de este proyecto en lenguaje COBOL ha sido necesaria la lectura y consulta de los manuales de referencia que se enumeran a continuación, casi todos ellos de IBM. También ha sido de gran utilidad los ejemplos de la página Web ESCOBOL que se menciona dentro del listado.

- IBM-Consulta de SQL - IBM DB2 Universal Database – Versión 7
- IBM-Programming guide - Enterprise COBOL for x/OS and OS/390 – Version 3 Release 2
- IBM-Languaje Rederence - Entreprise COBOL for z/OS and OS/390 – Version 3 Release 2
- IBM-Programming Guide – VisualAge COBOL – Version 3.02
- IBM-Language Reference – COBOL for OS/390 &VM, COBOL Set for AIX, VisualAge COBOL
- IBM-Installing and Configuring OS/390 Components for Remote ECD – Version 3.07
- IBM-Remote Edit-Compile-Debug User`s Guide – Version 3.01
- IBM-Getting Started COBOL– Version 3.02
- Oracle ProCOBOL™ Getting Started for Windows NT and Windows 95 – Release 8.0 June 1997
- ESCOBOL – www.escobol.com – Año 2011-2012